1. Lab 0 - Introduction to Module Set
2. Lab 1 - Discrete and Continuous-Time Signals
3. Lab 2 - Discrete-Time Systems
4. Lab 3 - Frequency Analysis
5. Lab 4 - Sampling and Reconstruction
6. Lab 5a - Digital Filter Design (part 1)
7. Lab 5b - Digital Filter Design (part 2)
8. Lab 6a - Discrete Fourier Transform and FFT (part 1)
9. Lab 6b - Discrete Fourier Transform and FFT (part 2)
10. Lab 7a - Discrete-Time Random Processes (part 1)
11. Lab 7b - Discrete-Time Random Processes (part 2)
12. Lab 7c - Power Spectrum Estimation
13. Lab 8 - Number Representation and Quantization
14. Lab 9a - Speech Processing (part 1)
15. Lab 9b - Speech Processing (part 2)
16. Lab 10a - Image Processing (part 1)
17. Lab 10b - Image Processing (part 2)

Lab 0 - Introduction to Module Set

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

These modules are a reproduction of a set of laboratory experiments developed for the course **ECE438 - Digital Signal Processing with Applications**, taught in the School of Electrical and Computer Engineering at Purdue University. This is a senior-level undergraduate course that covers the fundamentals of digital signal processing, along with several applications throughout the course. Some applications of particular emphasis are speech modeling, coding and synthesis, and also imaging processing topics including filtering, color spaces, halftoning, and tomography.

Laboratory experiments are performed each week during the semester, so these modules are designed to be completed in 2-4 hours. While the labs are performed as part of a lecture course, the lab modules contain most of the relevant background theory along with the lab exercises.

All of the lab exercises in this module set are written for Matlab by MathWorks. The modules are written for a student who has essentially no Matlab experience. A Matlab introduction is contained in the first module, and further Matlab skills and tips are provided as the modules progress forward.

External Links

Purdue University

School of Electrical and Computer Engineering

ECE438 Laboratory

Prof. Charles A. Bouman

Prof. Jan P. Allebach

Prof. Michael D. Zoltowski

Prof. Ilya Pollak

Lab 1 - Discrete and Continuous-Time Signals

Question and Comments

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

The purpose of this lab is to illustrate the properties of continuous and discrete-time signals using digital computers and the Matlab software environment. A continuous-time signal takes on a value at every point in time, whereas a discrete-time signal is only defined at integer values of the "time" variable. However, while discrete-time signals can be easily stored and processed on a computer, it is impossible to store the values of a continuous-time signal for all points along a segment of the real line. In later labs, we will see that digital computers are actually restricted to the storage of quantized discrete-time signals. Such signals are appropriately known as digital signals.

How then do we process continuous-time signals? In this lab, we will show that continuous-time signals may be processed by first approximating them by discrete-time signals using a process known as sampling. We will see that proper selection of the spacing between samples is crucial for an efficient and accurate approximation of a continuous-time signal. Excessively close spacing will lead to too much data, whereas excessively distant spacing will lead to a poor approximation of the continuous-time signal. Sampling will be an important topic in future labs, but for now we will use sampling to approximately compute some simple attributes of both real and synthetic signals.

Note: Be sure to read the guidelines for the written reports.

Matlab Review

Practically all lab tasks in the ECE438 lab will be performed using Matlab. Matlab (MATrix LABoratory) is a technical computing environment for numerical analysis, matrix computation, signal processing, and graphics. In this section, we will review some of its basic functions. For a short tutorial and some Matlab examples click here.

Starting Matlab and Getting Help

You can start Matlab (version 7.0) on your workstation by typing the command

```
matlab
```

in a command window. After starting up, you will get a Matlab window. To get help on any specific command, such as “plot”, you can type the following

```
help plot
```

in the “Command Window” portion of the Matlab window. You can do a keyword search for commands related to a topic by using the following.

```
lookfor topic
```

You can get an interactive help window using the function

```
helpdesk
```

or by following the Help menu near the top of the window.

Matrices and Operations

Every element in Matlab is a matrix. So, for example, the Matlab command

$a = [1 \quad 2 \quad 3]$

creates a matrix named "a" with dimensions of 1×3. The variable "a" is stored in what is called the Matlab workspace. The operation

$b = a$.'

stores the transpose of "a" into the vector "b". In this case, "b" is a 3×1 vector.

Since each element in Matlab is a matrix, the operation

$c = a*b$

computes the matrix product of "a" and "b" to generate a scalar value for "c" of $14 = 1*1 + 2*2 + 3*3$.

Often, you may want to apply an operation to each element of a vector. For example, you many want to square each value of "a". In this case, you may use the following command.

$c = a$.* a

The dot before the `*` tells Matlab that the multiplication should be applied to each corresponding element of "a". Therefore the `.*` operation is **not** a matrix operation. The dot convention works with many other Matlab commands such as divide `./` , and power `.^`. An error results if you try to perform element-wise operations on matrices that aren't the same size.

Note also that while the operation a .' performs a transpose on the matrix "a", the operation a' performs a **conjugate** transpose on "a" (transposes the matrix and conjugates each number in the matrix).

Matlab Scripts and Functions

Matlab has two methods for saving sequences of commands as standard files. These two methods are called **scripts** and **functions**. Scripts execute a

sequence of Matlab commands just as if you typed them directly into the Matlab command window. Functions differ from scripts in that they accept inputs and return outputs, and variables defined within a function are generally local to that function.

A script-file is a text file with the filename extension ".m". The file should contain a sequence of Matlab commands. The script-file can be run by typing its name at the Matlab prompt without the .m extension. This is equivalent to typing in the commands at the prompt. Within the script-file, you can access variables you defined earlier in Matlab. All variables in the script-file are global, i.e. after the execution of the script-file, you can access its variables at the Matlab prompt. For more help on scripts click here.

To create a function called `func`, you first create a text file called `func.m`. The first line of the file must be

```
function output = func(input)
```

where `input` designates the set of input variables, and `output` are your output variables. The rest of the function file then contains the desired operations. All variables within the function are local; that means the function cannot access Matlab workspace variables that you don't pass as inputs. After the execution of the function, you cannot access internal variables of the function. For more help on functions click here.

Continuous-Time Vs. Discrete-Time

The "Introduction" mentioned the important issue of representing continuous-time signals on a computer. In the following sections, we will illustrate the process of **sampling**, and demonstrate the importance of the **sampling interval** to the precision of numerical computations.

Analytical Calculation

Compute these two integrals. Do the computations manually.

1. **Equation:**

$$\int_0^{2\pi} sin^2\,(5t)dt$$

2. **Equation:**

$$\int_0^1 e^t dt$$

Note: Hand in your calculations of these two integrals. Show all work.

Displaying Continuous-Time and Discrete-Time Signals in Matlab

For help on the following topics, visit the corresponding link: Plot Function, Stem Command, and Subplot Command.

It is common to graph a discrete-time signal as dots in a Cartesian coordinate system. This can be done in the Matlab environment by using the `stem` command. We will also use the `subplot` command to put multiple plots on a single figure.

Start Matlab on your workstation and type the following sequence of commands.

```
n = 0:2:60;
y = sin(n/6);
subplot(3,1,1)
stem(n,y)
```

This plot shows the discrete-time signal formed by computing the values of the function sin $(t/6)$ at points which are uniformly spaced at intervals of

size 2. Notice that while $\sin(t/6)$ is a continuous-time function, the sampled version of the signal, $\sin(n/6)$, is a discrete-time function.

A digital computer cannot store all points of a continuous-time signal since this would require an infinite amount of memory. It is, however, possible to plot a signal which **looks like** a continuous-time signal, by computing the value of the signal at closely spaced points in time, and then connecting the plotted points with lines. The Matlab `plot` function may be used to generate such plots.

Use the following sequence of commands to generate two continuous-time plots of the signal $\sin(t/6)$.

```
n1 = 0:2:60;
z = sin(n1/6);
subplot(3,1,2)
plot(n1,z)
n2 = 0:10:60;
w = sin(n2/6);
subplot(3,1,3)
plot(n2,w)
```

As you can see, it is important to have many points to make the signal appear smooth. But how many points are enough for numerical calculations? In the following sections we will examine the effect of the sampling interval on the accuracy of computations.

Note: Submit a hard copy of the plots of the discrete-time function and two continuous-time functions. Label them with the

`title`

command, and include your names. Comment on the accuracy of each of the continuous time plots.

Numerical Computation of Continuous-Time Signals

For help on the following topics, click the corresponding link: MatLab Scripts, MatLab Functions, and the Subplot Command.

Background on Numerical Integration

One common calculation on continuous-time signals is integration. [link] illustrates a method used for computing the widely used Riemann integral. The Riemann integral approximates the area under a curve by breaking the region into many rectangles and summing their areas. Each rectangle is chosen to have the same width Δt, and the height of each rectangle is the value of the function at the start of the rectangle's interval.

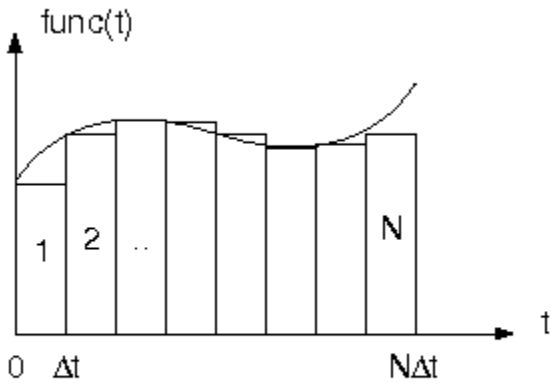


Illustration of the Riemann integral

To see the effects of using a different number of points to represent a continuous-time signal, write a Matlab function for numerically computing the integral of the function $sin^2(5t)$ over the interval $[0, 2\pi]$. The syntax of the function should be `I=integ1(N)` where I is the result and N is the

number of rectangles used to approximate the integral. This function should use the `sum` command and it should **not** contain any **for** loops!

Note: Since Matlab is an **interpreted** language, **for loops** are relatively slow. Therefore, we will avoid using loops whenever possible.

Next write an m-file script that evaluates $I(N)$ for $1 \leq N \leq 100$, stores the result in a vector and plots the resulting vector as a function of N. This m-file script may contain **for** loops.

Repeat this procedure for a second function `J=integ2(N)` which numerically computes the integral of exp (t) on the interval $[0, 1]$.

Note: Submit plots of $I(N)$ and $J(N)$ versus N. Use the subplot command to put both plots on a single sheet of paper. Also submit your Matlab code for each function. Compare your results to the analytical solutions from the "Analytical Calculation" section. Explain why $I(5) = I(10) = 0$.

Processing of Speech Signals

For this section download the speech.au file. For instructions on how to load and play audio signals click here.

Digital signal processing is widely used in speech processing for applications ranging from speech compression and transmission, to speech recognition and speaker identification. This exercise will introduce the process of reading and manipulating a speech signal.

First download the speech audio file speech.au, and then do the following:

1. Use the `auread` command to load the file **speech.au** into Matlab.
2. Plot the signal on the screen as if it were a continuous-time signal (i.e. use the `plot` command).
3. Play the signal via the digital-to-analog converter in your workstation with the Matlab `sound` function.

Note: Submit your plot of the speech signal.

Attributes of Continuous-Time Signals

For this section download the signal1.p function.

In this section you will practice writing .m-files to calculate the basic attributes of continuous-time signals. Download the function signal1.p. This is a pre-parsed pseudo-code file (P-file), which is a "pre-compiled" form of the Matlab function **signal1.m**. To evaluate this function, simply type `y = signal1(t)` where `t` is a vector containing values of time. Note that this Matlab function is valid for any real-valued time, `t` , so `y = signal1(t)` yields samples of a continuous-time function.

First plot the function using the `plot` command. Experiment with different values for the sampling period and the starting and ending times, and choose values that yield an accurate representation of the signal. Be sure to show the corresponding times in your plot using a command similar to `plot(t,y)`.

Next write individual Matlab functions to compute the minimum, maximum, and approximate energy of this particular signal. Each of these functions should just accept an input vector of times, `t`, and should call `signal1(t)` within the body of the function. You may use the built-in Matlab functions `min` and `max`. Again, you will need to experiment with the sampling period, and the starting and ending times so that your computations of the min, max, and energy are accurate.

Remember the definition of the energy is
Equation:

$$\text{energy} = \int_{-\infty}^{\infty} |\text{signal1}(t)|^2 dt \ .$$

Note: Submit a plot of the function, and the computed values of the min, max, and energy. Explain your choice of the sampling period, and the starting and ending times. Also, submit the code for your energy function.

Special Functions

Plot the following two continuous-time functions over the specified intervals. Write separate script files if you prefer. Use the `subplot` command to put both plots in a single figure, and be sure to label the time axes.

- $\text{sinc}(t)$ for t in $[-10\pi, 10\pi]$
- $\text{rect}(t)$ for t in $[-2, 2]$

Note: The function

```
rect(t)
```

may be computed in Matlab by using a Boolean expression. For example, if

```
t=-10:0.1:10
```

, then $y = rect(t)$ may be computed using the Matlab command

```
y=(abs(t)<=0.5)
```

.

Write an .m-script file to stem the following discrete-time function for $a = 0.8$, $a = 1.0$ and $a = 1.5$. Use the `subplot` command to put all three plots in a single figure. Issue the command `orient('tall')` just prior to printing to prevent crowding of the subplots.

- $a^n \left(u\left(n\right) - u\left(n - 10\right)\right)$ for n in $[-20, 20]$

Repeat this procedure for the function

- $\cos\left(\omega n\right) a^n u\left(n\right)$ for $\omega = \pi/4$, and n in $[-1, 10]$

Note: The unit step function $\mathrm{y} = \mathrm{u}(\mathrm{n})$ may be computed in Matlab using the command

```
y = (n>=0)
```

, where

```
n
```

is a vector of time indices.

Note: Submit all three figures, for a total of 8 plots. Also submit the printouts of your Matlab .m-files.

Sampling

The word **sampling** refers to the conversion of a continuous-time signal into a discrete-time signal. The signal is converted by taking its value, or sample, at uniformly spaced points in time. The time between two consecutive samples is called the **sampling period**. For example, a sampling period of 0.1 seconds implies that the value of the signal is stored every 0.1 seconds.

Consider the signal $f(t) = \sin(2\pi t)$. We may form a discrete-time signal, $x(n)$, by sampling this signal with a period of T_s. In this case,
Equation:

$$x(n) = f(T_s n) = \sin(2\pi T_s n) \ .$$

Use the `stem` command to plot the function $f(T_s n)$ defined above for the following values of T_s and n. Use the `subplot` command to put all the plots in a single figure, and scale the plots properly with the `axis` command.

1. $T_s = 1/10$, $0 \leq n \leq 100$; axis([0,100,-1,1])
2. $T_s = 1/3$, $0 \leq n \leq 30$; axis([0,30,-1,1])
3. $T_s = 1/2$, $0 \leq n \leq 20$; axis([0,20,-1,1])
4. $T_s = 10/9$, $0 \leq n \leq 9$; axis([0,9,-1,1])

Note: Submit a hardcopy of the figure containing all four subplots. Discuss your results. How does the sampled version of the signal with $T_s = 1/10$ compare to those with $T_s = 1/3$, $T_s = 1/2$ and $T_s = 10/9$?

Random Signals

For help on the Matlab random function, click here.

The objective of this section is to show how two signals that “look” similar can be distinguished by computing their average over a large interval. This

type of technique is used in signal demodulators to distinguish between the digits “1” and “0”.

Generate two discrete-time signals called “sig1” and “sig2” of length 1,000. The samples of “sig1” should be independent, Gaussian random variables with mean 0 and variance 1. The samples of “sig2” should be independent, Gaussian random variables with mean 0.2 and variance 1. Use the Matlab command `random` or `randn` to generate these signals, and then plot them on a single figure using the `subplot` command. (Recall that an alternative name for a Gaussian random variable is a **normal** random variable.)

Next form a new signal “ave1(n)” of length 1,000 such that “ave1(n)” is the average of the vector “sig1(1:n)” (the expression `sig1(1:n)` returns a vector containing the first n elements of “sig1”). Similarly, compute “ave2(n)” as the average of “sig2(1:n)”. Plot the signals “ave1(n)” and “ave2(n)” versus “n” on a single plot. Refer to help on the Matlab plot command for information on plotting multiple signals.

Note: Submit your plot of the two signals “sig1” and “sig2”. Also submit your plot of the two signals “ave1” and “ave2”. Comment on how the average values changes with n. Also comment on how the average values can be used to distinguish between random noise with different means.

2-D Signals

For help on the following topics, click the corresponding link: Meshgrid Command, Mesh Command, and Displaying Images.

So far we have only considered 1-D signals such as speech signals. However, 2-D signals are also very important in digital signal processing. For example, the elevation at each point on a map, or the color at each point on a photograph are examples of important 2-D signals. As in the 1-D case, we may distinguish between continuous-space and discrete-space signals.

However in this section, we will restrict attention to discrete-space 2-D signals.

When working with 2-D signals, we may choose to visualize them as images or as 2-D surfaces in a 3-D space. To demonstrate the differences between these two approaches, we will use two different display techniques in Matlab. Do the following:

1. Use the `meshgrid` command to generate the discrete-space 2-D signal
   Equation:

$$f(m,n) = 255|\text{sinc}(0.2m)\ \sin\ (0.2n)|$$

   for $-50 \leq m \leq 50$ and $-50 \leq n \leq 50$. See the help on meshgrid if you're unfamiliar with its usage.
2. Use the `mesh` command to display the signal as a surface plot.
3. Display the signal as an image. Use the command `colormap(gray(256))` just after issuing the `image` command to obtain a grayscale image. Read the help on image for more information.

Note: Hand in hardcopies of your mesh plot and image. For which applications do you think the surface plot works better? When would you prefer the image?

Lab 2 - Discrete-Time Systems

Question or Comments
Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

A discrete-time system is anything that takes a discrete-time signal as input and generates a discrete-time signal as output.[footnote] The concept of a system is very general. It may be used to model the response of an audio equalizer or the performance of the US economy.
A more general **behavioral** view of systems is anything that imposes constraints on a set of signals.

In electrical engineering, **continuous-time** signals are usually processed by electrical circuits described by differential equations. For example, any circuit of resistors, capacitors and inductors can be analyzed using mesh analysis to yield a system of differential equations. The voltages and currents in the circuit may then be computed by solving the equations.

The processing of **discrete-time** signals is performed by discrete-time systems. Similar to the continuous-time case, we may represent a discrete-time system either by a set of difference equations or by a block diagram of its implementation. For example, consider the following difference equation.
Equation:

$$y(n) = y(n-1) + x(n) + x(n-1) + x(n-2)$$

This equation represents a discrete-time **system**. It operates on the input signal $x(n)$ to produce the output signal $y(n)$. This system may also be defined by a system diagram as in [link].

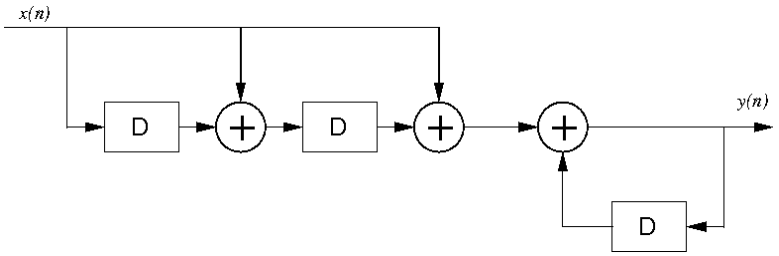


Diagram of a discrete-time system. (D = unit delay)

Mathematically, we use the notation $y = S[x]$ to denote a discrete-time system S with input signal $x(n)$ and output signal $y(n)$. Notice that the input and output to the system are the complete signals for all time **n**. This is important since the output at a particular time can be a function of past, present and future values of $x(n)$.

It is usually quite straightforward to write a computer program to implement a discrete-time system from its difference equation. In fact, programmable computers are one of the easiest and most cost effective ways of implementing discrete-time systems.

While equation [link] is an example of a linear time-invariant system, other discrete-time systems may be nonlinear and/or time varying. In order to understand discrete-time systems, it is important to first understand their classification into categories of linear/nonlinear, time-invariant/time-varying, causal/noncausal, memoryless/with-memory, and stable/unstable. Then it is possible to study the properties of restricted classes of systems, such as discrete-time systems which are linear, time-invariant and stable.

Background Exercises

Note: Submit these background exercises with the lab report.

Example Discrete-time Systems

Discrete-time digital systems are often used in place of analog processing systems. Common examples are the replacement of photographs with digital images, and conventional NTSC TV with direct broadcast digital TV. These digital systems can provide higher quality and/or lower cost through the use of standardized, high-volume digital processors.

The following two continuous-time systems are commonly used in electrical engineering:
Equation:

$$\text{differentiator:} \quad y(t) = \frac{d}{dt}x(t)$$

$$\text{integrator:} \quad y(t) = \int_{-\infty}^{t} x(\tau)d\tau$$

For each of these two systems, do the following:

- **i.**Formulate a discrete-time system that approximates the continuous-time function.
- **ii.**Write down the difference equation that describes your discrete-time system. Your difference equation should be in closed form, i.e. no summations.
- **iii.**Draw a block diagram of your discrete-time system as in [link].

Stock Market Example

One reason that digital signal processing (DSP) techniques are so powerful is that they can be used for very different kinds of signals. While most continuous-time systems only process voltage and current signals, a computer can process discrete-time signals which are essentially just sequences of numbers. Therefore DSP may be used in a very wide range of applications. Let's look at an example.

A stockbroker wants to see whether the average value of a certain stock is increasing or decreasing. To do this, the daily fluctuations of the stock values must be eliminated. A popular business magazine recommends three possible methods for computing this average.

Equation:

$$\text{avgvalue(today)} = \frac{1}{3}(\text{value(today)}+\text{value(yesterday)}+\text{value(2 days ago)})$$

Equation:

$$\text{avgvalue(today)} = 0.8*\text{avgvalue(yesterday)} + 0.2*(\text{value(today)})$$

Equation:

$$\text{avgvalue(today)} = \text{avgvalue(yesterday)} + \tfrac{1}{3}(\text{value(today)-(value(3 days ago))}$$

Do the following:

- For each of the these three methods: 1) write a difference equation, 2) draw a system diagram, and 3) calculate the impulse response.

- Explain why methods [link] and [link] are known as moving averages.

Example Discrete-Time Systems

Write two Matlab functions that will apply the differentiator and integrator systems, designed in the "Example Discrete-time Systems" section, to arbitrary input signals. Then apply the differentiator and integrator to the following two signals for $-10 \leq n \leq 20$.

- $\delta(n) - \delta(n-5)$
- $u(n) - u(n-(N+1))$ with $N = 10$

Hint: To compute the function $u(n)$ for $-10 \leq n \leq 20$, first set `n = -10:20`, and then use the Boolean expression `u = (n>=0)`.

For each of the four cases, use the **subplot** and **stem** commands to plot each input and output signal on a single figure.

Note: Submit printouts of your Matlab code and hardcopies containing the input and output signals. Discuss the stability of these systems.

Difference Equations

In this section, we will study the effect of two discrete-time filters. The first filter, $y = S_1[x]$, obeys the difference equation
Equation:

$$y(n) = x(n) - x(n-1)$$

and the second filter, $y = S_2[x]$, obeys the difference equation
Equation:

$$y(n) = \frac{1}{2} y(n-1) + x(n)$$

Write Matlab functions to implement each of these filters.

Note: In Matlab, when implementing a difference equation using a loop structure, it is very good practice to pre-define your output vector before entering into the loop. Otherwise, Matlab has to resize the output vector at each iteration. For example, say you are using a FOR loop to filter the signal $x(n)$, yielding an output $y(n)$. You can pre-define the output vector by issuing the command

```
y=zeros(1,N)
```

before entering the loop, where

```
N
```

is the final length of

```
y
```

. For long signals, this speeds up the computation dramatically.

Now use these functions to calculate the impulse response of each of the following 5 systems: S_1, S_2, $S_1\,(S_2)$ (i.e., the series connection with S_1 following S_2), $S_2\,(S_1)$ (i.e., the series connection with S_2 following S_1), and $S_1 + S_2$.

Note: For each of the five systems, draw and submit a system diagram (use only delays, multiplications and additions as in [link]). Also submit plots of each impulse response. Discuss your observations.

Audio Filtering

For this section download the [missing_resource: music.au] click here.

Use the command `auread` to load the file [missing_resource: music.au]`sound` to listen to the signal.

Next filter the audio signal with each of the two systems S_1 and S_2 from the previous section. Listen to the two filtered signals.

Note: How do the filters change the sound of the audio signals? Explain your observations.

Inverse Systems

Consider the system $y = S_2\,[x]$ from the "Difference Equations" section. Find a difference equation for a new system $y = S_3\,[x]$ such that $\delta = S_3\,[S_2\,[\delta]]$ where δ denotes the discrete-time impulse function $\delta(n)$. Since both systems S_2 and S_3 are LTI, the time-invariance and superposition properties can be used to obtain $x = S_3\,[S_2\,[x]]$ for **any** discrete-time signal x. We say that the systems S_3 and S_2 are inverse filters because they cancel out the effects of each other.

Hint: The system $y = S_3\,[x]$ can be described by the difference equation
Equation:

$$y(n) = ax(n) \; + \; bx(n-1)$$

where a and b are constants.

Write a Matlab function `y = S3(x)` which implements the system S_3. Then obtain the impulse response of both S_3 and $S_3[S_2\,[\delta]]$.

Note: Draw a system diagram for the system S_3, and submit plots of the impulse responses for S_3 and $S_3(S_2)$.

System Tests

For this section download the zip file bbox.zip.

Often it is necessary to determine if a system is linear and/or time-invariant. If the inner workings of a system are not known, this task is impossible because the linearity and time-invariance properties must hold true for all possible inputs signals. However, it is possible to show that a system is non-linear or time-varying because only a single instance must be found where the properties are violated.

The zip file bbox.zip contains three "black-box" systems in the files bbox1.p, bbox2.p, and bbox3.p. These files work as Matlab functions, with the syntax `y=bboxN(x)`, where `x` and `y` are the input and the output signals, and `N = 1, 2 or 3`. Exactly one

of these systems is non-linear, and exactly one of them is time-varying. Your task is to find the non-linear system and the time-varying system.
Hints:

1. You should try a variety of input signals until you find a counter-example.
2. When testing for time-invariance, you need to look at the responses to a signal and to its delayed version. Since all your signals in MATLAB have finite duration, you should be very careful about shifting signals. In particular, if you want to shift a signal x by M samples to the left, x should start with at least M zeros. If you want to shift x by M samples to the right, x should end with at least M zeros.
3. When testing for linearity, you may find that simple inputs such as the unit impulse do not accomplish the task. In this case, you should try something more complicated like a sinusoid or a random signal generated with the `random` command.

Note: State which system is non-linear, and which system is time-varying. Submit plots of input/output signal pairs that support your conclusions. Indicate on the plots why they support your conclusions.

Stock Market Example

For this section download stockrates.mat. For help on loading Matlab files click here.

Load stockrates.mat into Matlab. This file contains a vector, called **rate**, of daily stock market exchange rates for a publicly traded stock.

Apply filters [link] and [link] from the "Stock Market Example" section of the background exercises to smooth the stock values. When you apply the filter of [link] you will need to initialize the value of $\mathrm{avgvalue}(\mathrm{yesterday})$. Use an initial value of 0. Similarly, in [link], set the initial values of the "value" vector to 0 (for the days prior to the start of data collection). Use the `subplot` command to plot the original stock values, the result of filtering with [link], and the result of filtering with [link].

Note: Submit your plots of the original and filtered exchange-rates. Discuss the advantages and disadvantages of the two filters. Can you suggest a better method for initializing the filter outputs?

Lab 3 - Frequency Analysis

Questions and Comments
Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

In this experiment,we will use Fourier series and Fourier transforms to analyze continuous-time and discrete-time signals and systems. The Fourier representations of signals involve the decomposition of the signal in terms of complex exponential functions. These decompositions are very important in the analysis of linear time-invariant (LTI) systems, due to the property that the response of an LTI system to a complex exponential input is a complex exponential of the same frequency! Only the amplitude and phase of the input signal are changed. Therefore, studying the frequency response of an LTI system gives complete insight into its behavior.

In this experiment and others to follow, we will use the Simulink extension to Matlab. Simulink is an icon-driven dynamic simulation package that allows the user to represent a system or a process by a block diagram. Once the representation is completed, Simulink may be used to digitally simulate the behavior of the continuous or discrete-time system. Simulink inputs can be Matlab variables from the workspace, or waveforms or sequences generated by Simulink itself. These Simulink-generated inputs can represent continuous-time or discrete-time sources. The behavior of the simulated system can be monitored using Simulink's version of common lab instruments, such as scopes, spectrum analyzers and network analyzers.

Background Exercises

Note: Submit these background exercises with the lab report.

Synthesis of Periodic Signals

Each signal given below represents one period of a periodic signal with period T_0.

1. Period $T_0 = 2$. For $t \in [0, 2]$:
   Equation:

$$s(t) = \text{rect}\left(t - \frac{1}{2}\right)$$

2. Period $T_0 = 1$. For $t \in \left[-\frac{1}{2}, \frac{1}{2}\right]$:
   Equation:

$$s(t) = rect(2t) - \frac{1}{2}$$

For each of these two signals, do the following:

- **i**Compute the Fourier series expansion in the form
  Equation:

$$s(t) = a_0 + \sum_{k=1}^{\infty} A_k \sin(2\pi k f_0 t + \theta_k)$$

  where $f_0 = 1/T_0$.

> **Note:** You may want to use one of the following references: Sec. 4.1 of Digital Signal Processing", by Proakis and Manolakis, 1996; Sec. 4.2 of Signals and Systems", by Oppenheim and Willsky, 1983; Sec. 3.3 of Signals and Systems", Oppenheim and Willsky, 1997. Note that in the expression above, the function in the summation is $\sin(2\pi k f_0 t + \theta_k)$, rather than a complex sinusoid. The formulas in the above references must be modified to accommodate this. You can

compute the cos/sin version of the Fourier series, then convert the coefficients.

- **ii**Sketch the signal on the interval $[0, T_0]$.

Magnitude and Phase of Discrete-Time Systems

For the discrete-time system described by the following difference equation,
Equation:

$$y(n) = 0.9y(n-1) + 0.3x(n) + 0.24x(n-1)$$

- **i**Compute the impulse response.
- **ii**Draw a system diagram.
- **iii**Take the Z-transform of the difference equation using the linearity and the time shifting properties of the Z-transform.
- **iv**Find the transfer function, defined as
  Equation:

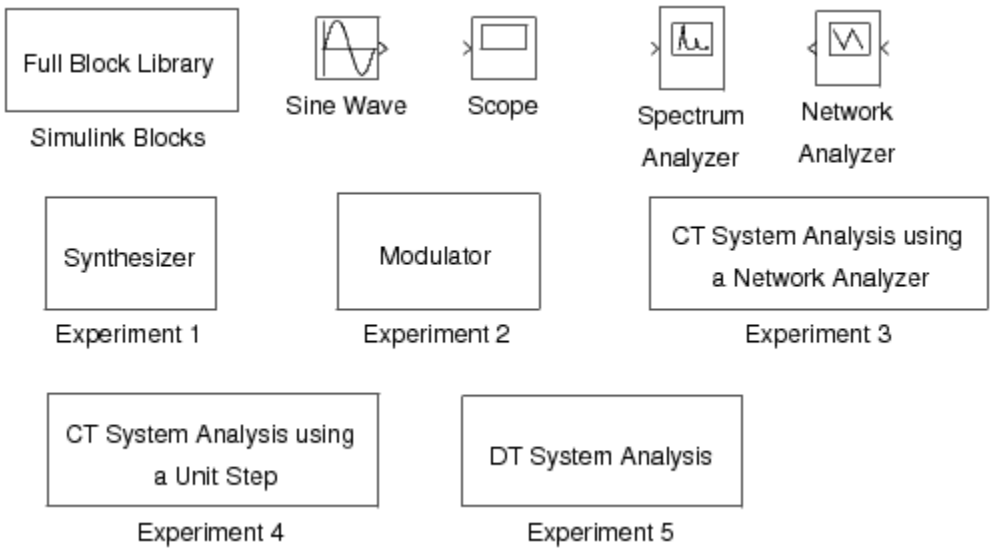
$$H(z) \equiv \frac{Y(z)}{X(z)}$$

- **v**Use Matlab to compute and plot the magnitude and phase responses, $\left|H\left(e^{j\omega}\right)\right|$ and $\angle H\left(e^{j\omega}\right)$, for $-\pi < \omega < \pi$. You may use Matlab commands `phase` and `abs`.

Getting Started with Simulink

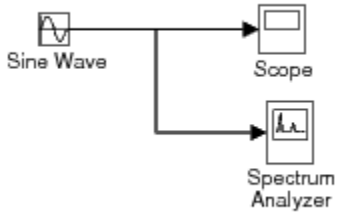
In this section, we will learn the basics of Simulink and build a simple system.

For help on "Simulink" click here. For the following sections download the file Lab3Utilities.zip.



Simulink utilities for lab 3.

To get the library of Simulink functions for this laboratory, download the file Lab3Utilities.zip. Once Matlab is started, type “Lab3” to bring up the library of Simulink components shown in [link]. This library contains a full library of Simulink blocks, a spectrum analyzer and network analyzer designed for this laboratory, a sine wave generator, a scope, and pre-design systems for each of the experiments that you will be running.



Simulink model for the introductory example.

In order to familiarize yourself with Simulink, you will first build the system shown in [link]. This system consists of a sine wave generator that feeds a scope and a spectrum analyzer.

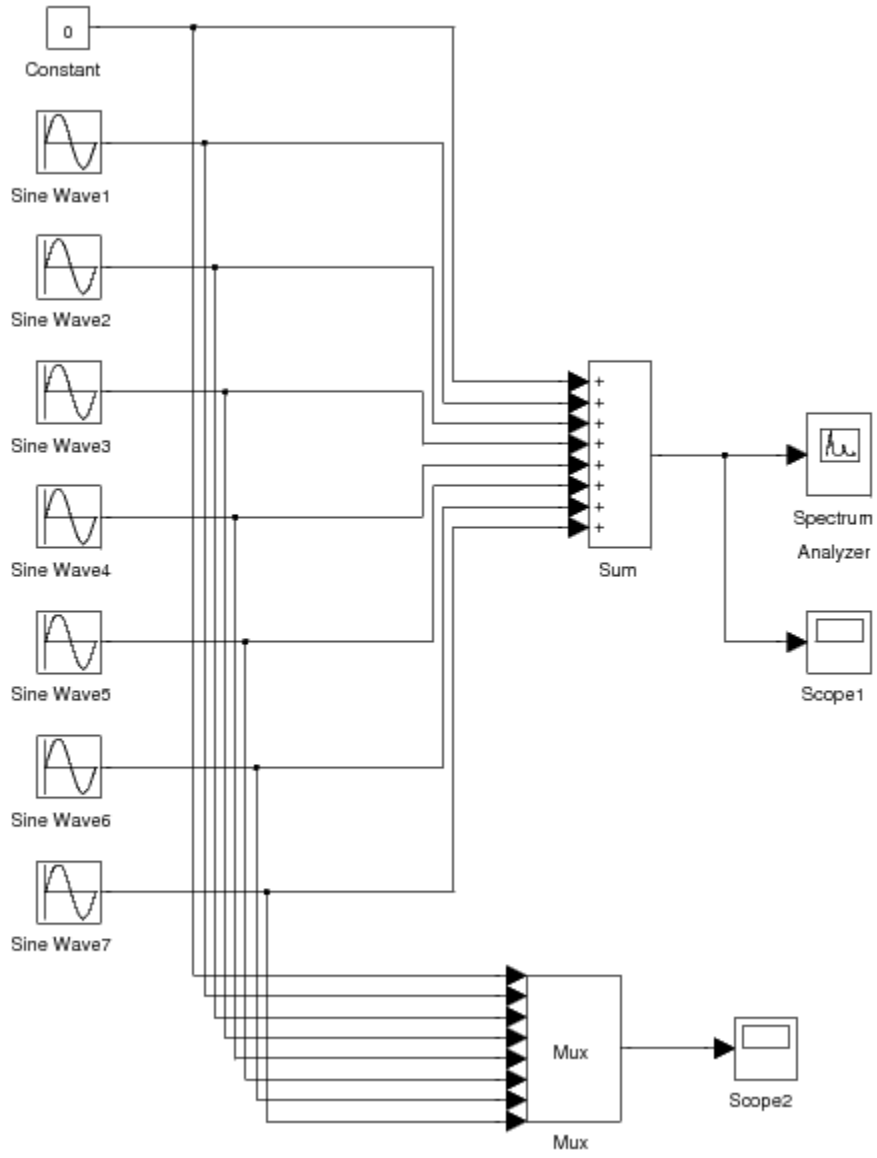
1. Open a window for a new system by using the `New` option from the `File` pull-down menu, and select `Model`.
2. Drag the `Sine Wave`, `Scope`, and `Spectrum Analyzer` blocks from the `Lab3` window into the new window you created.
3. Now you need to connect these three blocks. With the left mouse button, click on the output of the `Sine Wave` and drag it to the input of the `Scope`. Now use the right button to click on the line you just created, and drag to the input of the `Spectrum Analyzer` block. Your system should now look like [link].
4. Double click on the `Scope` block to make the plotting window for the scope appear.
5. Set the simulation parameters by selecting `Configuration Parameters` from the `Simulation` pull-down menu. Under the `Solver` tab, set the `Stop time` to 50, and the `Max step size` to 0.02. Then select `OK`. This will allow the Spectrum Analyzer to make a more accurate calculation.
6. Start the simulation by using the `Start` option from the `Simulation` pull-down menu. A standard Matlab figure window will pop up showing the output of the `Spectrum Analyzer`.
7. Change the frequency of the sine wave to `5*pi rad/sec` by double clicking on the `Sine Wave` icon and changing the number in the `Frequency` field. Restart the simulation. Observe the change in the waveform and its spectral density. If you want to change the time scaling in the plot generated by the spectrum analyzer, from the Matlab prompt use the `subplot(2,1,1)` and `axis()` commands.
8. When you are done, close the system window you created by using the `Close` option from the `File` pull-down menu.

Continuous-Time Frequency Analysis

For help on the following topics select the corresponding link: simulink or printing figures in Simulink.

In this section, we will study the use and properties of the continuous-time Fourier transform with Simulink. The Simulink package is especially useful for continuous-time systems because it allows the simulation of their behavior on a digital computer.

Synthesis of Periodic Signals



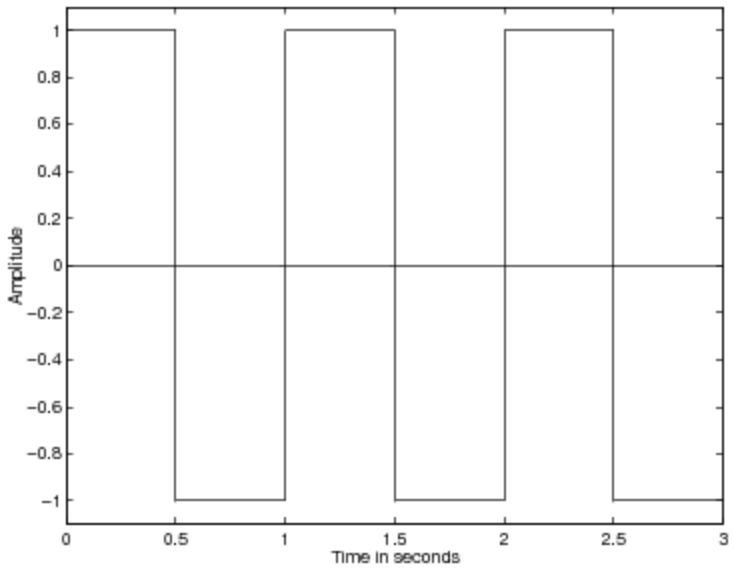
Simulink model for the synthesizer experiment.

Double click the icon labeled `Synthesizer` to bring up a model as shown in [link]. This system may be used to synthesize periodic signals by adding together the harmonic components of a Fourier series expansion. Each `Sin Wave` block can be set to a specific frequency, amplitude and phase. The initial settings of the `Sin Wave` blocks are set to generate the Fourier series expansion

Equation:

$$x\left(t\right)=0+\sum_{\substack{k=1\\k\,\mathrm{odd}}}^{13}\frac{4}{k\pi}\,\sin\left(2\pi kt\right)\ .$$

These are the first 8 terms in the Fourier series of the periodic square wave shown in [link].



The desired waveform for the synthesizer experiment.

Run the model by selecting `Start` under the `Simulation` menu. A graph will pop up that shows the synthesized square wave signal and its spectrum. This is the output of the `Spectrum Analyzer`. After the simulation runs for a while, the `Spectrum Analyzer` element will update the plot of the spectral energy and the incoming waveform. Notice that the energy is concentrated in peaks corresponding to the individual sine waves. Print the output of the `Spectrum Analyzer`.

You may have a closer look at the synthesized signal by double clicking on the `Scope1` icon. You can also see a plot of all the individual sine waves by double clicking on the `Scope2` icon.

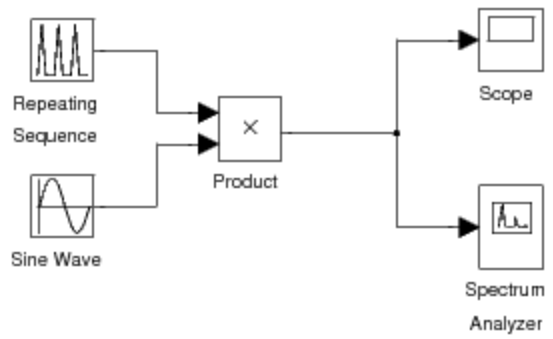
Synthesize the two periodic waveforms defined in the "Synthesis of Periodic Signals" section of the background exercises. Do this by setting the frequency, amplitude, and phase of each sinewave generator to the proper values. For each case, print the output of the `Spectrum Analyzer`.

Note: Hand in plots of the

`Spectrum Analyzer`

output for each of the three synthesized waveforms. For each case, comment on how the synthesized waveform differs from the desired signal, and on the structure of the spectral density.

Modulation Property



Simulink model for the modulation experiment.

Double click the icon labeled `Modulator` to bring up a system as shown in [link]. This system modulates a triangular pulse signal with a sine wave. You can control the duration and duty cycle of the triangular envelope and the frequency of the modulating sine wave. The system also contains a spectrum analyzer which plots the modulated signal and its spectrum.

Generate the following signals by adjusting the `Time values` and `Output values` of the `Repeating Sequence` block and the `Frequency` of the `Sine Wave`. The `Time values` vector contains entries spanning one period of the repeating signal. The `Output values` vector contains the values of the repeating signal at the times specified in the `Time values` vector. Note that the `Repeating Sequence` block does NOT create a discrete time signal. It creates a continuous time signal by connecting the output values with line segments. Print the output of the `Spectrum Analyzer` for each signal.

1. Triangular pulse duration of 1 sec; period of 2 sec; modulating frequency of 10 Hz (initial settings of the experiment).
2. Triangular pulse duration of 1 sec; period of 2 sec; modulating frequency of 15 Hz.
3. Triangular pulse duration of 1 sec; period of 3 sec; modulating frequency of 10 Hz.
4. Triangular pulse duration of 1 sec; period of 6 sec; modulating frequency of 10 Hz.

Notice that the spectrum of the modulated signal consists of of a comb of impulses in the frequency domain, arranged around a center frequency.

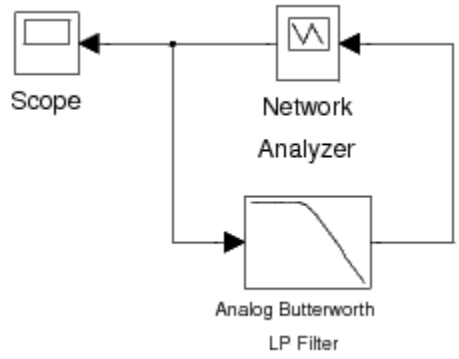
Note: Hand in plots of the output of the

`Spectrum Analyzer`

for each signal. Answer following questions: 1) What effect does changing the modulating frequency have on the spectral density? 2) Why does the spectrum have a comb structure and what is the spectral distance between

impulses? Why? 3) What would happen to the spectral density if the period of the triangle pulse were to increase toward infinity? (in the limit)

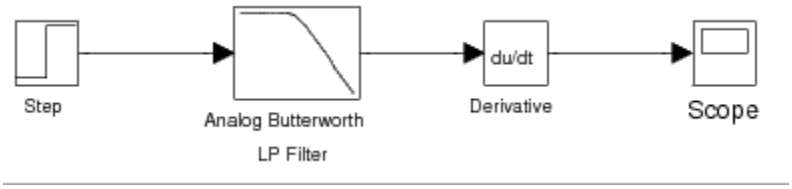
System Analysis



Simulink model for the continuous-time system analysis experiment using a network analyzer.

Double click the icon labeled `CT System Analysis using a Network Analyzer` to bring up a system as shown in [link]. This system includes a `Network Analyzer` model for measuring the frequency response of a system. The `Network Analyzer` works by generating a weighted chirp signal (shown on the `Scope`) as an input to the system-under-test. The analyzer measures the frequency response of the input and output of the system and computes the transfer function. By computing the inverse Fourier transform, it then computes the impulse response of the system. Use this setup to compute the frequency and impulse response of the given fourth order Butterworth filter with a cut-off frequency of 1Hz. Print the figure showing the magnitude response, the

phase response and the impulse response of the system. To use the tall mode to obtain a larger printout, type `orient('tall');` directly before you print.



Simulink model for the continuous-time system analysis experiment using a unit step.

An alternative method for computing the impulse response is to input a step into the system and then to compute the derivative of the output. The model for doing this is given in the `CT System Analysis using a Unit Step` block. Double click on this icon and compute the impulse response of the filter using this setup ([link]). Make sure that the characteristics of the filter are the same as in the previous setup. After running the simulation, print the graph of the impulse response.

Note: Hand in the printout of the output of the

`Network Analyzer`

(magnitude and phase of the frequency response, and the impulse response) and the plot of the impulse response obtained using a unit step. What are the advantages and disadvantages of each method?

Discrete-Time Frequency Analysis

In this section of the laboratory, we will study the use of the discrete-time Fourier transform.

Discrete-Time Fourier Transform

The DTFT (Discrete-Time Fourier Transform) is the Fourier representation used for finite energy discrete-time signals. For a discrete-time signal, $x(n)$, we denote the DTFT as the function $X\left(e^{j\omega}\right)$ given by the expression
Equation:

$$X\left(e^{j\omega}\right) = \sum_{n=-\infty}^{\infty} x\left(n\right)e^{-j\omega n} \ .$$

Since $X\left(e^{\mathrm{j}\omega}\right)$ is a periodic function of ω with a period of 2π, we need only to compute $X\left(e^{\mathrm{j}\omega}\right)$ for $-\pi < \omega < \pi$.

Write a Matlab function `X=DTFT(x,n0,dw)` that computes the DTFT of the discrete-time signal `x` Here `n0` is the time index corresponding to the 1st element of the `x` vector, and `dw` is the spacing between the samples of the Matlab vector `X`. For example, if `x` is a vector of length `N`, then its DTFT is computed by
Equation:

$$X\left(w\right) = \sum_{n=1}^{N} x\left(n\right)e^{-jw(n+n0-1)}$$

where w is a vector of values formed by `w=(-pi:dw:pi)` .

Note: In Matlab,

`j`

or

`i`

is defined as $\sqrt{-1}$. However, you may also compute this value using the Matlab expression

```
i=sqrt(-1)
```

.

For the following signals use your DTFT function to

- **i**Compute $X\left(e^{j\omega}\right)$
- **ii**Plot the magnitude and the phase of $X\left(e^{j\omega}\right)$ in a single plot using the `subplot` command.

Note: Use the

`abs()`

and

`angle()`

commands.

1. **Equation:**

$$x(n) = \delta(n)$$

2. **Equation:**

$$x(n) = \delta(n-5)$$

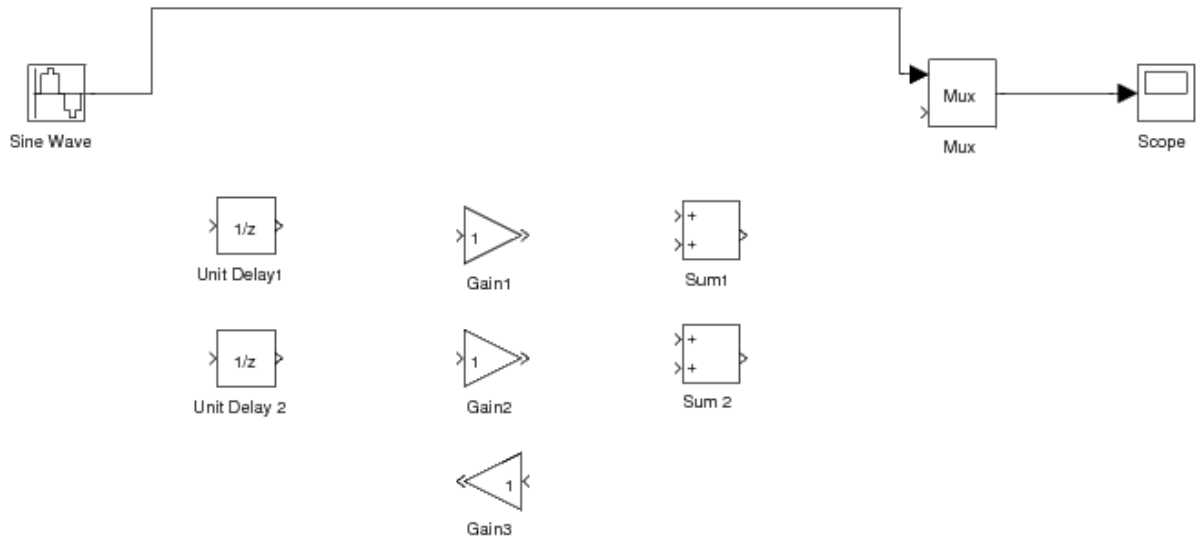
3. **Equation:**

$$x\left(n\right) = \left(0.5\right)^{n} u\left(n\right)$$

Note: Hand in a printout of your Matlab function. Also hand in plots of the DTFT's magnitude and phase for each of the three signals.

System Analysis

For help on printing Simulink system windows click here.



Incomplete Simulink setup for the discrete-time system analysis experiment.

Double click the icon labeled `DT System Analysis` to bring up an incomplete block diagram as shown in [link]. It is for a model that takes a discrete-time sine signal, processes it according to a difference equation and plots the multiplexed input and output signals in a graph window. Complete this block diagram such that it implements the following difference equation given in "Magnitude and Phase of Discrete-Time Systems" of the background exercises.

Equation:

$$y(n) = 0.9y(n-1) + 0.3x(n) + 0.24x(n-1)$$

You are provided with the framework of the setup and the building blocks that you will need. You can change the values of the `Gain` blocks by double clicking on them. After you complete the setup, adjust the frequency of `Sine Wave` to the following frequencies: $\omega = \pi/16$, $\omega = \pi/8$, and $\omega = \pi/4$. For each frequency, make magnitude response measurements using the input and output sequences shown in the graph window. Compare your measurements with the values of the magnitude response $\left|H\left(e^{j\omega}\right)\right|$ which you computed in the background exercises at these frequencies.

An alternative way of finding the frequency response is taking the DTFT of the impulse response. Use your DTFT function to find the frequency response of this system from its impulse response. The impulse response was calculated in "Magnitude and Phase of Discrete-Time Systems" of the background exercises. Plot the impulse response, and the magnitude and phase of the frequency response in the same figure using the `subplot` command.

Note: Hand in the following: 1) Printout of your completed block diagram. 2) Table of both the amplitude measurements you made and their theoretical values. 3) Printout of the figure with the impulse response, and the magnitude and phase of the frequency response.

Lab 4 - Sampling and Reconstruction

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

It is often desired to analyze and process continuous-time signals using a computer. However, in order to process a continuous-time signal, it must first be digitized. This means that the continuous-time signal must be sampled and quantized, forming a digital signal that can be stored in a computer. Analog systems can be converted to their discrete-time counterparts, and these digital systems then process discrete-time signals to produce discrete-time outputs. The digital output can then be converted back to an analog signal, or **reconstructed**, through a digital-to-analog converter. [link] illustrates an example, containing the three general components described above: a sampling system, a digital signal processor, and a reconstruction system.

When designing such a system, it is essential to understand the effects of the sampling and reconstruction processes. Sampling and reconstruction may lead to different types of distortion, including low-pass filtering, aliasing, and quantization. The system designer must insure that these distortions are below acceptable levels, or are compensated through additional processing.

Example of a typical digital signal processing system.

Sampling Overview

Sampling is simply the process of measuring the value of a continuous-time signal at certain instants of time. Typically, these measurements are uniformly separated by the sampling period, T_s. If $x(t)$ is the input signal, then the sampled signal, $y(n)$, is as follows:
Equation:

$$y\left(n\right) = x(t)|_{t=nT_s} \ .$$

A critical question is the following: What sampling period, T_s, is required to accurately represent the signal $x(t)$? To answer this question, we need to look at the frequency domain representations of $y(n)$ and $x(t)$. Since $y(n)$ is a discrete-time signal, we represent its frequency content with the discrete-time Fourier transform (DTFT), $Y\left(e^{j\omega}\right)$. However, $x(t)$ is a continuous-time signal, requiring the use of the continuous-time Fourier transform (CTFT), denoted as $X(f)$. Fortunately, $Y\left(e^{j\omega}\right)$ can be written in terms of $X(f)$:
Equation:

$$\begin{aligned} Y\left(e^{j\omega}\right) &= \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X(f)|_{f=\frac{\omega-2\pi k}{2\pi T_s}} \\ &= \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X\left(\frac{\omega-2\pi k}{2\pi T_s}\right) \ . \end{aligned}$$

Consistent with the properties of the DTFT, $Y\left(e^{j\omega}\right)$ is periodic with a period 2π. It is formed by rescaling the amplitude and frequency of $X(f)$, and then repeating it in frequency every 2π. The critical issue of the relationship in [link] is the frequency content of $X(f)$. If $X(f)$ has frequency components that are above $1/(2T_s)$, the repetition in frequency will cause these components to overlap with (i.e. add to) the components below $1/(2T_s)$. This causes an unrecoverable distortion, known as **aliasing**, that will prevent a perfect reconstruction of $X(f)$. We will illustrate this

later in the lab. The $1/(2T_s)$ “cutoff frequency” is known as the **Nyquist frequency**.

To prevent aliasing, most sampling systems first low pass filter the incoming signal to ensure that its frequency content is below the Nyquist frequency. In this case, $Y\left(e^{j\omega}\right)$ can be related to $X(f)$ through the $k = 0$ term in [link]:
Equation:

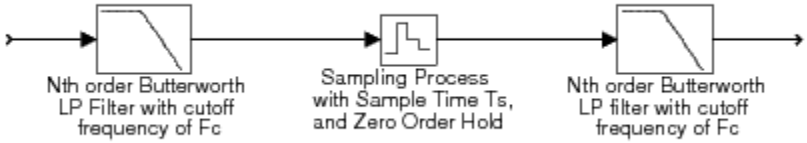
$$Y\left(e^{j\omega}\right) = \frac{1}{T_s} X\left(\frac{\omega}{2\pi T_s}\right) \quad \text{for } \omega \in [-\pi, \pi] \ .$$

Here, it is understood that $Y\left(e^{j\omega}\right)$ is periodic with period 2π. Note in this expression that $Y\left(e^{j\omega}\right)$ and $X(f)$ are related by a simple scaling of the frequency and magnitude axes. Also note that $\omega = \pi$ in $Y\left(e^{j\omega}\right)$ corresponds to the Nyquist frequency, $f = 1/(2T_s)$ in $X(f)$.

Sometimes after the sampled signal has been digitally processed, it must then converted back to an analog signal. Theoretically, this can be done by converting the discrete-time signal to a sequence of continuous-time impulses that are weighted by the sample values. If this continuous-time “impulse train” is filtered with an ideal low pass filter, with a cutoff frequency equal to the Nyquist frequency, a scaled version of the original low pass filtered signal will result. The spectrum of the reconstructed signal $S(f)$ is given by
Equation:

$$S\left(f\right) = \begin{cases} Y\left(e^{j2\pi f T_s}\right) & \text{for } |f| < \frac{1}{2T_s} \\ 0 & \text{otherwise.} \end{cases}$$

Sampling and Reconstruction Using Sample-and-Hold



Sampling and reconstruction using a sample-and-hold.

In practice, signals are reconstructed using digital-to-analog converters. These devices work by reading the current sample, and generating a corresponding output voltage for a period of T_s seconds. The combined effect of sampling and D/A conversion may be thought of as a single sample-and-hold device. Unfortunately, the sample-and-hold process distorts the frequency spectrum of the reconstructed signal. In this section, we will analyze the effects of using a zero$^{\text{th}}$ $-$ order sample-and-hold in a sampling and reconstruction system. Later in the laboratory, we will see how the distortion introduced by a sample-and-hold process may be reduced through the use of discrete-time interpolation.

[link] illustrates a system with a low-pass input filter, a sample-and-hold device, and a low-pass output filter. If there were no sampling, this system would simply be two analog filters in cascade. We know the frequency response for this simpler system. Any differences between this and the frequency response for the entire system is a result of the sampling and reconstruction. Our goal is to compare the two frequency responses using Matlab. For this analysis, we will assume that the filters are N^{th} order Butterworth filters with a cutoff frequency of f_c, and that the sample-and-hold runs at a sampling rate of $f_s = 1/T_s$.

We will start the analysis by first examining the ideal case. Consider replacing the sample-and-hold with an ideal impulse generator, and assume that instead of the Butterworth filters we use perfect low-pass filters with a cutoff of f_c . After analyzing this case we will modify the results to account for the sample-and-hold and Butterworth filter roll-off.

If an ideal impulse generator is used in place of the sample-and-hold, then the frequency spectrum of the impulse train can be computed by combining the sampling equation in [link] with the reconstruction equation in [link].
Equation:

$$\begin{aligned} S(f) &= Y\left(e^{j2\pi fT_s}\right) \\ &= \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X\left(\frac{2\pi fT_s - 2\pi k}{2\pi T_s}\right) \\ &= \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X(f - kf_s), \text{ for } |f| \leq \frac{1}{2T_s} \ . \\ S(f) &= 0 \text{ for } |f| > \frac{1}{2T_s} \ . \end{aligned}$$

If we assume that $f_s > 2f_c$, then the infinite sum reduces to one term. In this case, the reconstructed signal is given by
Equation:

$$S\left(f\right) = \frac{1}{T_s}X(f) \ .$$

Notice that the reconstructed signal is scaled by the factor $\frac{1}{T_s}$.

Of course, the sample-and-hold does not generate perfect impulses. Instead it generates a pulse of width T_s, and magnitude equal to the input sample. Therefore, the new signal out of the sample-and-hold is equivalent to the old signal (an impulse train) convolved with the pulse
Equation:

$$p\left(t\right) = \text{rect}\left(\frac{t}{T_s} - \frac{1}{2}\right) \ .$$

Convolution in the time domain is equivalent to multiplication in the frequency domain, so this convolution with $p(t)$ is equivalent to multiplying by the Fourier transform $P(f)$ where
Equation:

$$|P(f)| = T_s \left|\text{sinc}\left(f/f_s\right)\right| \ .$$

Finally, the magnitude of the frequency response of the N-th order Butterworth filter is given by
Equation:

$$\left|H_b(f)\right| = \frac{1}{1+\left(\frac{f}{f_c}\right)^N} \ .$$

We may calculate the complete magnitude response of the sample-and-hold system by combining the effects of the Butterworth filters in [link], the ideal sampling system in [link], and the sample-and-hold pulse width in [link]. This yields the final expression
Equation:

$$\begin{aligned} |H(f)| &= \left|H_b(f)P(f)\frac{1}{T_s}H_b(f)\right| \\ &= \left(\frac{1}{1+\left(\frac{f}{f_c}\right)^N}\right)^2 \left|\text{sinc}\left(f/f_s\right)\right| \ . \end{aligned}$$

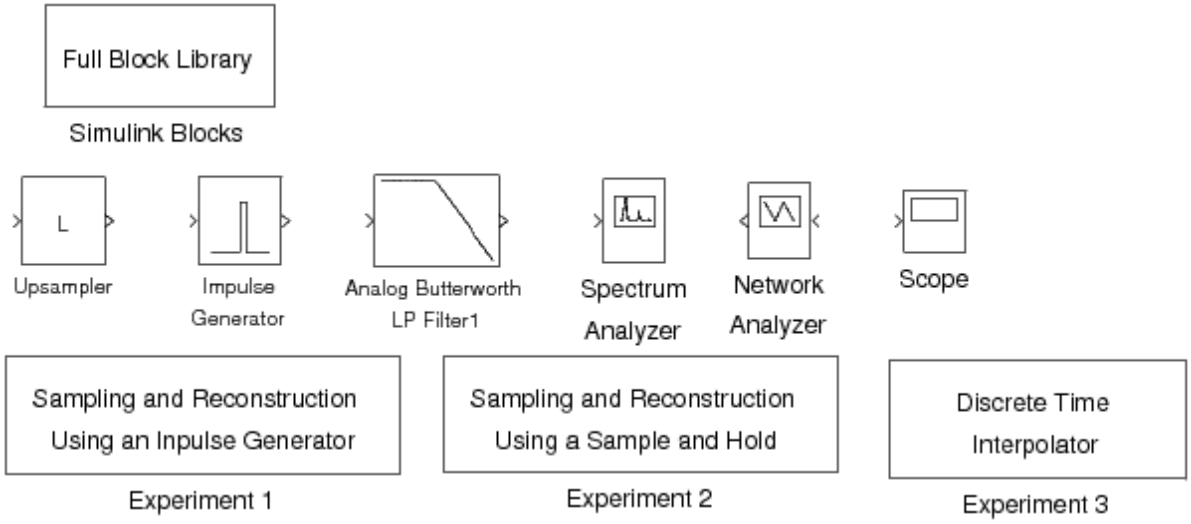
Notice that the expression $|\text{sinc}(f/f_s)|$ produces a roll-off in frequency which will attenuate frequencies close to the Nyquist rate. Generally, this roll-off is not desirable.

INLAB REPORT

Do the following using $T_s = 1$ sec, $f_c = 0.45$ Hz, and $N = 20$. Use Matlab to produce the plots (magnitude only), for frequencies in the range: `f = -1:.001:1`.

- Compute and plot the magnitude response of the system in [link] without the sample-and-hold device.
- Compute and plot the magnitude response of the complete system in [link].
- Comment on the shape of the two magnitude responses. How might the magnitude response of the sample-and-hold affect the design considerations of a high quality audio CD player?

Simulink Overview



Simulink utilities for lab 4.

In this lab we will use Simulink to simulate the effects of the sampling and reconstruction processes. Simulink treats all signals as **continuous-time** signals. This means that "sampled" signals are really just continuous-time signals that contain a series of finite-width pulses. The height of each of

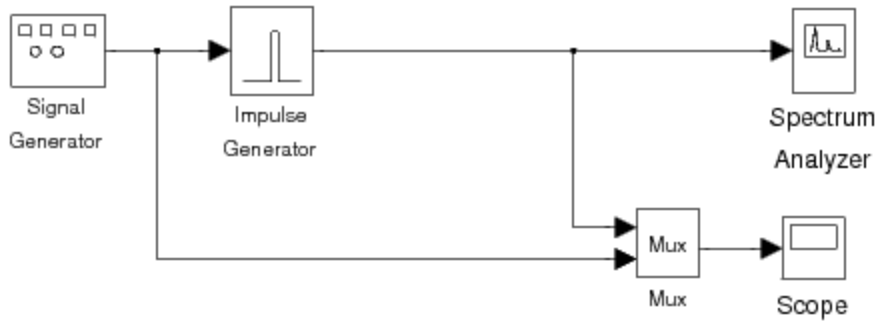
these pulses is the amplitude of the input signal at the beginning of the pulse. In other words, both the sampling action and the zero-order-hold reconstruction are done at the same time; the discrete-time signal itself is never generated. This means that the impulse-generator block is really a “pulse-generator”, or zero-order-hold device. Remember that, in Simulink, frequency spectra are computed on continuous-time signals. This is why many aliased components will appear in the spectra.

Sampling and Reconstruction Using An Impulse Generator

For help on the following topics select the corresponding links: simulink and printing figures in simulink. For the following section, download the file Lab4Utils.zip.

In this section, we will experiment with the sampling and reconstruction of signals using a pulse generator. This pulse generator is the combination of an ideal impulse generator and a perfect zero-order-hold device.

In order to run the experiment, first download the required Lab4Utilities. Once Matlab is started, type “Lab4”. A set of Simulink blocks and experiments will come up as shown in [link].



Simulink model for sampling and reconstruction using an impulse generator.

Before starting this experiment, use the MATLAB command `close all` to close all figures other than the Simulink windows. Double click on the icon named `Sampling and Reconstruction Using An Impulse Generator` to bring up the first experiment as shown in [link]. In this experiment, a sine wave is sampled at a frequency of 1 Hz; then the sampled discrete-time signal is used to generate rectangular impulses of duration 0.3 sec and amplitude equal to the sample values. The block named `Impulse Generator` carries out both the sampling of the sine wave and its reconstruction with pulses. A single `Scope` is used to plot both the input and output of the impulse generator, and a `Spectrum Analyzer` is used to plot the output pulse train and its spectrum.

First, run the simulation with the frequency of input sine wave set to 0. 1 Hz (initial setting of the experiment). Let the simulation run until it terminates to get an accurate plot of the output frequencies. Then print the output of `Scope` and the `Spectrum Analyzer`. Be sure to label your plots.

Note: Submit the plot of the input/output signals and the plot of the output signal and its frequency spectrum. On the plot of the spectrum of the reconstructed signal, circle the aliases, i.e. the components that do NOT correspond to the input sine wave.

Ideal impulse functions can only be approximated. In the initial setup, the pulse width is 0. 3 sec, which is less then the sampling period of 1 sec. Try setting the pulse width to 0. 1 sec and run the simulation. Print the output of the `Spectrum Analyzer`.

Note: Submit the plot of the output frequency spectrum for a pulse width of 0. 1 sec. Indicate on your plot what has changed and **explain why**.

Set the pulse width back to 0.3 sec and change the frequency of the sine wave to 0.8 Hz. Run the simulation and print the output of the `Scope` and the `Spectrum Analyzer`.

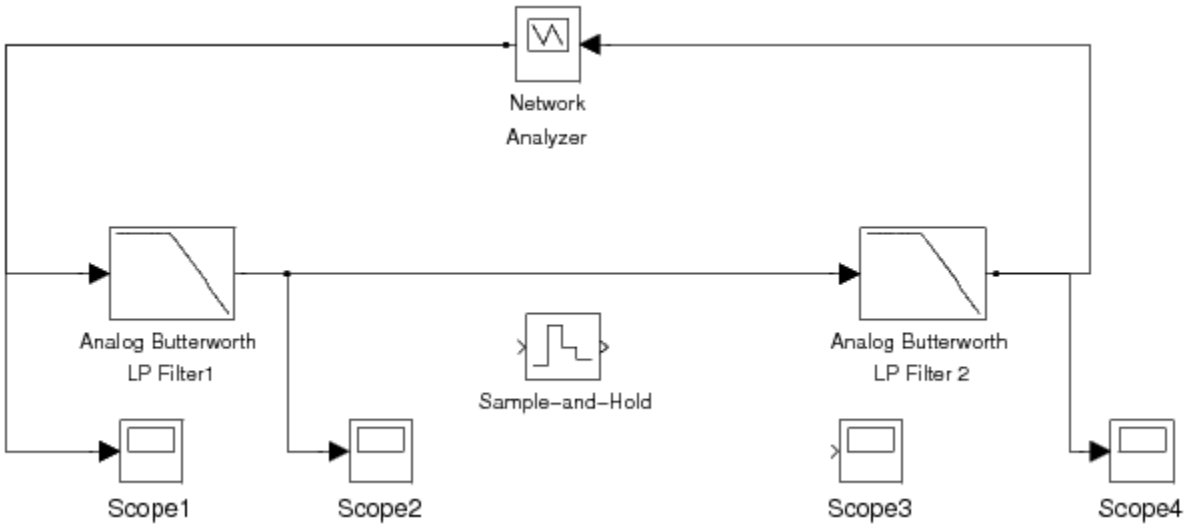
Note: Submit the plot of the input/output signals and the plot of the output signal and its frequency spectrum. On the frequency plot, label the frequency peak that corresponds to the lowest frequency (the fundamental component) of the output signal. Explain why the lowest frequency is no longer the same as the frequency of the input sinusoid.

Leave the input frequency at 0.8 Hz. Now insert a filter right after the impulse generator. Use a 10th order Butterworth filter with a cutoff frequency of 0.5 Hz. Connect the output of the filter to the `Spectrum Analyzer` and the `Mux`. Run the simulation, and print the output of `Scope` and the `Spectrum Analyzer`.

Note: Submit the plot of the input/output signals and the plot of the output signal and its frequency spectrum. Explain why the output signal has the observed frequency spectrum.

Sampling and Reconstruction with Sample and Hold

For help on printing figures in Simulink select the link.



Initial Simulink model for sampling and reconstruction using a sample-and-hold. This system only measures the frequency response of the analog filters.

In this section, we will sample a continuous-time signal using a sample-and-hold and then reconstruct it. We already know that a sample-and-hold followed by a low-pass filter does **not** result in perfect reconstruction. This is because a sample-and-hold acts like a pulse generator with a pulse duration of one sampling period. This “pulse shape” of the sample-and-hold is what distorts the frequency spectrum (see Section "Sampling and Reconstruction Using a Sample-and-Hold").

To start the second experiment, double click on the icon named `Sampling and Reconstruction Using A Sample and Hold`. [link] shows the initial setup for this exercise. It contains 4 `Scopes` to monitor the processing done in the sampling and reconstruction system. It also contains a `Network Analyzer` for measuring the frequency response and the impulse response of the system.

The `Network Analyzer` works by generating a weighted chirp signal (shown on `Scope 1`) as an input to the system-under-test. The frequency spectrum of this chirp signal is known. The analyzer then measures the

frequency content of the output signal (shown on `Scope 4`). The transfer function is formed by computing the ratio of the output frequency spectrum to the input spectrum. The inverse Fourier transform of this ratio, which is the impulse response of the system, is then computed.

In the initial setup, the `Sample-and-Hold` and `Scope 3` are not connected. There is no sampling in this system, just two cascaded low-pass filters. Run the simulation and observe the signals on the `Scopes`. Wait for the simulation to end.

Note: Submit the figure containing plots of the magnitude response, the phase response, and the impulse response of this system. Use the tall mode to obtain a larger printout by typing

```
orient('tall')
```

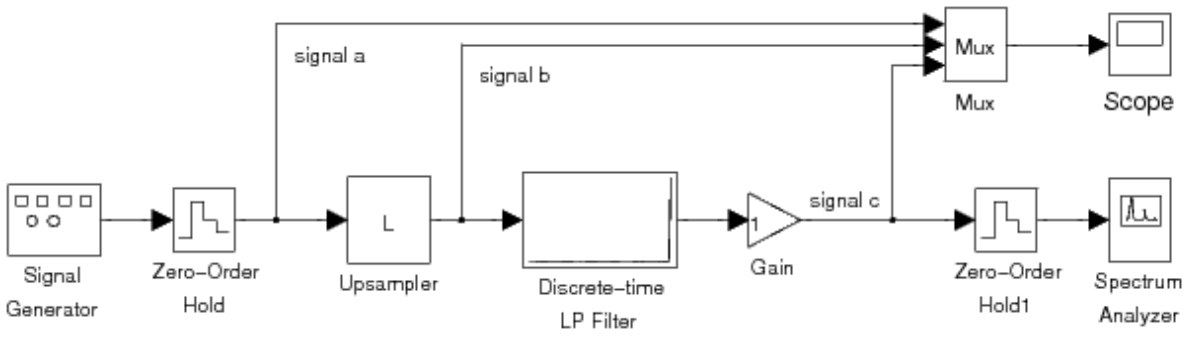
directly before you print.

Double-click the `Sample-and-Hold` and set its `Sample time` to 1. Now, insert the `Sample-and-Hold` in between the two filters and connect `Scope 3` to its output. Run the simulation and observe the signals on the `Scopes`.

Note: Submit the figure containing plots of the magnitude response, the phase response, and the impulse response of this system. Explain the reason for the difference in the shape of this magnitude response versus the previous magnitude response. Give an **analytical expression** for the behavior of the magnitude plot for frequencies below 0.45 Hz.

Discrete-Time Interpolation

For help on printing figures in Simulink select the link.



Simulink model for discrete-time interpolation.

In the previous experiments, we saw that the frequency content of a signal must be limited to half the sampling rate in order to avoid aliasing effects in the reconstructed signal. However, reconstruction can be difficult if the sampling rate is chosen to be just above the Nyquist frequency. Reconstruction is much easier for a higher sampling rate because the sampled signal will better "track" the original analog signal.

From another perspective, the analog output filter must have a very sharp cutoff in order to accurately reconstruct a signal that was sampled just above the Nyquist rate. Such filters are difficult and expensive to manufacture. Alternatively, a higher sampling rate allows the use analog output filters that have a slow roll-off. These filters are much less expensive. However, a high sampling rate is not practical in most applications, as it results in unnecessary samples and excessive storage requirements.

A practical solution to this dilemma is to **interpolate** the digital signal to create new (artificial) samples between the existing samples. This may be done by first upsampling the digital representation, and then filtering out unwanted components using a discrete-time filter. This discrete-time filter

serves the same purpose as an analog filter with a sharp cutoff, but it is generally simpler and more cost effective to implement.

Upsampling a signal by a factor of L is simply the process of inserting $L-1$ zeros in between each sample. The frequency domain relationship between a signal $x(n)$ and its upsampled version $z(n)$ can be shown to be the following
Equation:

$$Z\left(e^{j\omega}\right) = X\left(e^{j\omega L}\right) \ .$$

Therefore the DTFT of $z(n)$ is simply $X\left(e^{j\omega}\right)$ compressed in frequency by a factor of L. Since $X\left(e^{j\omega}\right)$ has a period of 2π, $Z\left(e^{j\omega}\right)$ will have a period of $2\pi/L$. All of the original information of $x(n)$ will be contained in the interval $[-\pi/L, \pi/L]$ of $Z\left(e^{j\omega}\right)$, and the new aliases that are created in the interval $[-\pi, \pi]$ are the unwanted components that need to be filtered out. In the time domain, this filtering has the effect of changing the inserted zeros into artificial samples of $x(n)$, commonly known as **interpolated** samples.

[link] shows a Simulink model that demonstrates discrete-time interpolation. The interpolating system contains three main components: an upsampler which inserts $L-1$ zeros between each input sample, a discrete-time low pass filter which removes aliased signal components in the interpolated signal, and a gain block to correct the magnitude of the final signal. Notice that "signal a" is the input discrete-time signal while "signal c" is the final interpolated discrete-time signal.

Open the experiment by double clicking on the icon labeled `Discrete Time Interpolator`. The components of the system are initially set to interpolate by a factor of 1. This means that the input and output signals will be the same except for a delay. Run this model with the initial settings, and observe the signals on the `Scope`.

Simulink represents any discrete-time signal by holding each sample value over a certain time period. This representation is equivalent to a sample-

and-hold reconstruction of the underlying discrete-time signal. Therefore, a continuous-time `Spectrum Analyzer` may be used to view the frequency content of the output "signal c". The `Zero-Order Hold` at the `Gain` output is required as a buffer for the `Spectrum Analyzer` in order to set its internal sampling period.

The lowest frequency component in the spectrum corresponds to the frequency content of the original input signal, while the higher frequencies are aliased components resulting from the sample-and-hold reconstruction. Notice that the aliased components of "signal c" appear at multiples of the sampling frequency of 1 Hz. Print the output of the `Spectrum Analyzer`.

Note: Submit your plot of "signal c" and its frequency spectrum. Circle the aliased components in your plot.

Next modify the system to upsample by a factor of 4 by setting this parameter in the `Upsampler`. You will also need to set the `Sample time` of the DT filter to 0.25. This effectively increases the sampling frequency of the system to 4 Hz. Run the simulation again and observe the behavior of the system. Notice that zeros have been inserted between samples of the input signal. After you get an accurate plot of the output frequency spectrum, print the output of the `Spectrum Analyzer`.

Notice the new aliased components generated by the upsampler. Some of these spectral components lie between the frequency of the original signal and the new sampling frequency, 4 Hz. These aliases are due to the zeros that are inserted by the upsampler.

Note: Submit your plot of "signal c" and its frequency spectrum. On your frequency plot, circle the first aliased component and label the value of its center frequency. Comment on the shape of the envelope of the spectrum.

Notice in the previous `Scope` output that the process of upsampling causes a decrease in the energy of the sample-and-hold representation by a factor of 4. This is the reason for using the `Gain` block.

Now determine the gain factor of the `Gain` block and the cutoff frequency of the `Discrete-time LP filter` needed to produce the desired interpolated signal. Run the simulation and observe the behavior of the system. After you get an accurate plot of the output frequency spectrum, print the output of the `Spectrum Analyzer`. Identify the change in the location of the aliased components in the output signal.

Note: Submit your plot of "signal c" and its frequency spectrum. Give the values of the cutoff frequency and gain that were used. On your frequency plot, circle the location of the first aliased component. Explain why discrete-time interpolation is desirable before reconstructing a sampled signal.

Discrete-Time Decimation

For the following section, download the file [missing_resource: music.au]how to load and play sudio signals select the link.

In the previous section, we used interpolation to increase the sampling rate of a discrete-time signal. However, we often have the opposite problem in which the desired sampling rate is lower than the sampling rate of the available data. In this case, we must use a process called **decimation** to reduce the sampling rate of the signal.

Decimating, or **downsampling**, a signal $x(n)$ by a factor of D is the process of creating a new signal $y(n)$ by taking only every D^{th} sample of $x(n)$. Therefore $y(n)$ is simply $x(Dn)$. The frequency domain relationship between $y(n)$ and $x(n)$ can be shown to be the following:
Equation:

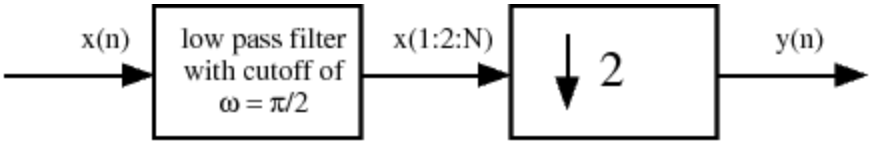
$$Y\left(e^{j\omega}\right) = \frac{1}{D}\sum_{k=0}^{D-1} X\left(\frac{\omega - 2\pi k}{D}\right) \ .$$

Notice the similarity of [link] to the sampling theorem equation in [link]. This similarity should be expected because decimation is the process of sampling a discrete-time signal. In this case, $Y\left(e^{j\omega}\right)$ is formed by taking $X\left(e^{j\omega}\right)$ in the interval $[-\pi, \pi]$ and expanding it in frequency by a factor of D. Then it is repeated in frequency every 2π, and scaled in amplitude by $1/D$. For similar reasons as described for equation [link], aliasing will be prevented if in the interval $[-\pi, \pi]$, $X\left(e^{j\omega}\right)$ is zero outside the interval $[-\pi/D, \pi/D]$. Then [link] simplifies to

Equation:

$$Y\left(e^{j\omega}\right) = \frac{1}{D} X\left(e^{j\frac{\omega}{D}}\right) \quad \text{for } \omega \in [-\pi, \pi] \ .$$

A system for decimating a signal is shown in [link]. The signal is first filtered using a low pass filter with a cutoff frequency of $\pi/2$ rad/sample. This insures that the signal is band limited so that the relationship in [link] holds. The output of the filter is then subsampled by removing every other sample.



This system decimates a discrete-time signal by a factor of 2.

For the following section download [missing_resource: music.au]`music.au` using `auread`, and then play it back with `sound`.

The signal contained in `music.au` was sampled at 16 kHz, so it will sound much too slow when played back at the default 8 kHz sampling rate.

To correct the sampling rate of the signal, form a new signal, sig1 , by selecting every other sample of the music vector. Play the new signal using `sound`, and listen carefully to the new signal.

Next compute a second subsampled signal, sig2 , by first low pass filtering the original music vector using a discrete-time filter of length 20, and with a cutoff frequency of $\pi/2$. Then decimate the filtered signal by 2, and listen carefully to the new signal.

Note: You can filter the signal by using the Matlab command

```
output = conv(s,h)
```

, where

`s`

is the signal, and

`h`

is the impulse response of the desired filter. To design a length M low-pass filter with cutoff frequency

`W`

rad/sample, use the command

```
h = fir1(M,W/pi)
```

.

Note: Hand in the Matlab code for this exercise. Also, comment on the quality of the audio signal generated by using the two decimation methods. Was there any noticeable distortion in **sig1**? If so, describe the distortion.

Lab 5a - Digital Filter Design (part 1)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This is the first part of a two week laboratory in digital filter design. The first week of the laboratory covers some basic examples of FIR and IIR filters, and then introduces the concepts of FIR filter design. Then the second week covers systematic methods of designing both FIR and IIR filters.

Background on Digital Filters

In digital signal processing applications, it is often necessary to change the relative amplitudes of frequency components or remove undesired frequencies of a signal. This process is called **filtering**. Digital filters are used in a variety of applications. In Laboratory 4, we saw that digital filters may be used in systems that perform interpolation and decimation on discrete-time signals. Digital filters are also used in audio systems that allow the listener to adjust the bass (low-frequency energy) and the treble (high frequency energy) of audio signals.

Digital filter design requires the use of both frequency domain and time domain techniques. This is because filter design specifications are often given in the frequency domain, but filters are usually implemented in the time domain with a difference equation. Typically, frequency domain analysis is done using the Z-transform and the discrete-time Fourier Transform (DTFT).

In general, a linear and time-invariant causal digital filter with input $x(n)$ and output $y(n)$ may be specified by its difference equation
Equation:

$$y(n) = \sum_{i=0}^{N-1} b_i x(n-i) - \sum_{k=1}^{M} a_k y(n-k)$$

where b_i and a_k are coefficients which parameterize the filter. This filter is said to have N zeros and M poles. Each new value of the output signal, $y(n)$, is determined by past values of the output, and by present and past values of the input. The impulse response, $h(n)$, is the response of the filter to an input of $\delta(n)$, and is therefore the solution to the recursive difference equation

Equation:

$$h(n) = \sum_{i=0}^{N-1} b_i \delta(n-i) - \sum_{k=1}^{M} a_k h(n-k) \ .$$

There are two general classes of digital filters: infinite impulse response (IIR) and finite impulse response (FIR). The FIR case occurs when $a_k = 0$, for all k. Such a filter is said to have no poles, only zeros. In this case, the difference equation in [link] becomes

Equation:

$$h(n) = \sum_{i=0}^{N-1} b_i \delta(n-i) \ .$$

Since [link] is no longer recursive, the impulse response has finite duration N.

In the case where $a_k \neq 0$, the difference equation usually represents an IIR filter. In this case, [link] will usually generate an impulse response which has non-zero values as $n \to \infty$. However, later we will see that for certain values of $a_k \neq 0$ and b_i, it is possible to generate an FIR filter response.

The Z-transform is the major tool used for analyzing the frequency response of filters and their difference equations. The Z-transform of a discrete-time

signal, $x(n)$, is given by
Equation:

$$X\left(z\right)=\sum_{n=-\infty}^{\infty}x\left(n\right)z^{-n}\ .$$

where z is a complex variable. The DTFT may be thought of as a special case of the Z-transform where z is evaluated on the unit circle in the complex plane.
Equation:

$$\begin{aligned}X\left(e^{j\omega}\right) &= X(z)|_{z=e^{j\omega}} \\ &= \sum_{n=-\infty}^{\infty}x\left(n\right)e^{-j\omega n}\end{aligned}$$

From the definition of the Z-transform, a change of variable $m=n-K$ shows that a delay of K samples in the time domain is equivalent to multiplication by z^{-K} in the Z-transform domain.
Equation:

$$\begin{aligned}x(n-K) &\overset{Z}{\leftrightarrow} \sum_{n=-\infty}^{\infty}x\left(n-K\right)z^{-n} \\ &= \sum_{m=-\infty}^{\infty}x\left(m\right)z^{-(m+K)} \\ &= z^{-K}\sum_{m=-\infty}^{\infty}x\left(m\right)z^{-m} \\ &= z^{-K}X\left(z\right)\end{aligned}$$

We may use this fact to re-write [link] in the Z-transform domain, by taking Z-transforms of both sides of the equation:
Equation:

$$Y\left(z\right) = \sum_{i=0}^{N-1} b_i z^{-i} X\left(z\right) - \sum_{k=1}^{M} a_k z^{-k} Y\left(z\right)$$

$$Y\left(z\right)\left(1 + \sum_{k=1}^{M} a_k z^{-k}\right) = X\left(z\right) \sum_{i=0}^{N-1} b_i z^{-i}$$

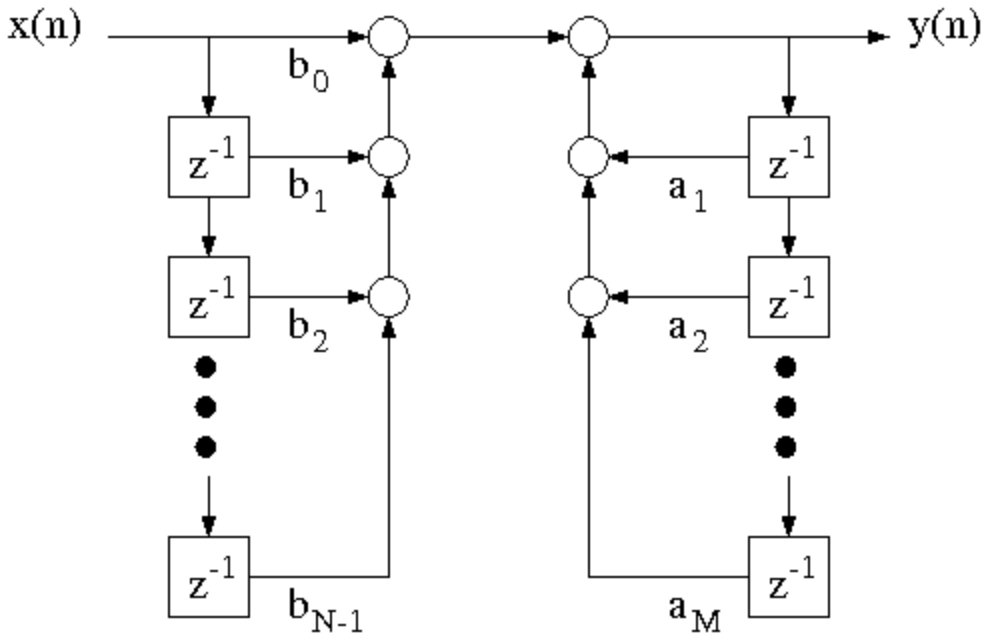
$$H\left(z\right) \triangleq \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N-1} b_i z^{-i}}{1 + \sum_{k=1}^{M} a_k z^{-k}}$$

From this formula, we see that any filter which can be represented by a linear difference equation with constant coefficients has a rational transfer function (i.e. a transfer function which is a ratio of polynomials). From this result, we may compute the frequency response of the filter by evaluating $H(z)$ on the unit circle:

Equation:

$$H\left(e^{j\omega}\right) = \frac{\sum_{i=0}^{N-1} b_i e^{-j\omega i}}{1 + \sum_{k=1}^{M} a_k e^{-j\omega k}} \ .$$

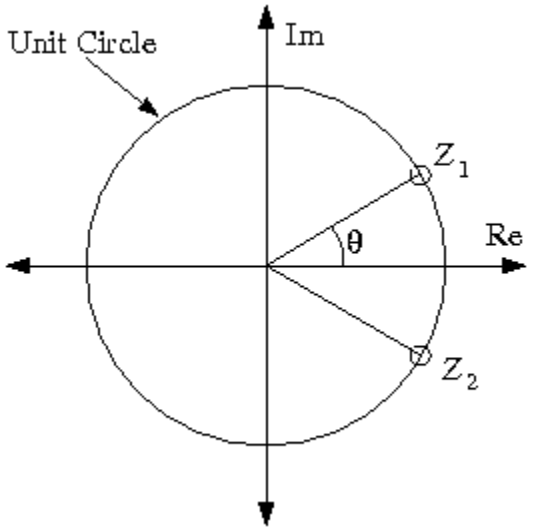
There are many different methods for implementing a general recursive difference equation of the form [link]. Depending on the application, some methods may be more robust to quantization error, require fewer multiplies or adds, or require less memory. [link] shows a system diagram known as the direct form implementation; it works for any discrete-time filter described by the difference equation in [link]. Note that the boxes containing the symbol z^{-1} represent unit delays, while a parameter written next to a signal path represents multiplication by that parameter.



Direct form implementation for a discrete-time filter described by a linear recursive difference equation.

Design of a Simple FIR Filter

Download the files, nspeech1.mat and DTFT.m for the following section.



Location of two zeros for simple a FIR filter.

To illustrate the use of zeros in filter design, you will design a simple second order FIR filter with the two zeros on the unit circle as shown in [link]. In order for the filter's impulse response to be real-valued, the two zeros must be complex conjugates of one another:
Equation:

$$z_1 = e^{j\theta}$$

Equation:

$$z_2 = e^{-j\theta}$$

where θ is the angle of z_1 relative to the positive real axis. We will see later that $\theta \in [0, \pi]$ may be interpreted as the location of the zeros in the frequency response.

The transfer function for this filter is given by
Equation:

$$\begin{aligned} H_f(z) &= \left(1 - z_1 z^{-1}\right)\left(1 - z_2 z^{-1}\right) \\ &= \left(1 - e^{j\theta} z^{-1}\right)\left(1 - e^{-j\theta} z^{-1}\right) \\ &= 1 - 2\cos\theta z^{-1} + z^{-2} \ . \end{aligned}$$

Use this transfer function to determine the difference equation for this filter. Then draw the corresponding system diagram and compute the filter's impulse response $h(n)$.

This filter is an FIR filter because it has impulse response $h(n)$ of finite duration. Any filter with only zeros and no poles other than those at 0 and $\pm\infty$ is an FIR filter. Zeros in the transfer function represent frequencies that are not passed through the filter. This can be useful for removing unwanted frequencies in a signal. The fact that $H_f(z)$ has zeros at $e^{\pm j\theta}$ implies that $H_f\left(e^{\pm j\theta}\right) = 0$. This means that the filter will not pass pure sine waves at a frequency of $\omega = \theta$.

Use Matlab to compute and plot the magnitude of the filter's frequency response $\left|H_f\left(e^{j\omega}\right)\right|$ as a function of ω on the interval $-\pi < \omega < \pi$, for the following three values of θ:

- **i)**
  Equation:

  $$\theta = \pi/6$$

- **ii)**
  Equation:

  $$\theta = \pi/3$$

- **ii)**
  Equation:

  $$\theta = \pi/2$$

Put all three plots on the same figure using the `subplot` command.

Note: Submit the difference equation, system diagram, and the analytical expression of the impulse response for the filter $H_f(z)$. Also submit the plot of the magnitude response for the three values of θ. Explain how the value of θ affects the magnitude of the filter's frequency response.

In the next experiment, we will use the filter $H_f(z)$ to remove an undesirable sinusoidal interference from a speech signal. To run the experiment, first download the audio signal nspeech1.mat, and the M-file DTFT.m Load `nspeech1.mat` into Matlab using the command `load nspeech1`. This will load the signal `nspeech1` into the workspace. Play `nspeech1` using the `sound` command, and then plot 101 samples of the signal for the time indices (100:200).

We will next use the `DTFT` command to compute samples of the DTFT of the audio signal. The `DTFT` command has the syntax

```
[X,w]=DTFT(x,M)
```

where `x` is a signal which is assumed to start at time $n = 0$, and M specifies the number of output points of the DTFT. The command `[X,w]=DTFT(x,0)` will generate a DTFT that is the same duration as the input; if this is not sufficient, it may be increased by specifying `M` . The outputs of the function are a vector `X` containing the samples of the DTFT, and a vector `w` containing the corresponding frequencies of these samples.

Compute the magnitude of the DTFT of 1001 samples of the audio signal for the time indices (100:1100). Plot the magnitude of the DTFT samples versus frequency for $|\omega| < \pi$. Notice that there are two large peaks corresponding to the sinusoidal interference signal. Use the Matlab `max` command to determine the exact frequency of the peaks. This will be the value of θ that we will use for filtering with $H_f(z)$.

Note: Use the command

```
[Xmax,Imax]=max(abs(X))
```

to find the value and index of the maximum element in

`X`

. θ can be derived using this index.

Write a Matlab function `FIRfilter(x)` that implements the filter $H_f(z)$ with the measured value of θ and outputs the filtered signal (Hint: Use convolution). Apply the new function `FIRfilter` to the `nspeech1` vector to attenuate the sinusoidal interference. Listen to the filtered signal to hear the effects of the filter. Plot 101 samples of the signal for the time indices (100:200), and plot the magnitude of the DTFT of 1001 samples of the filtered signal for the time indices (100:1100).

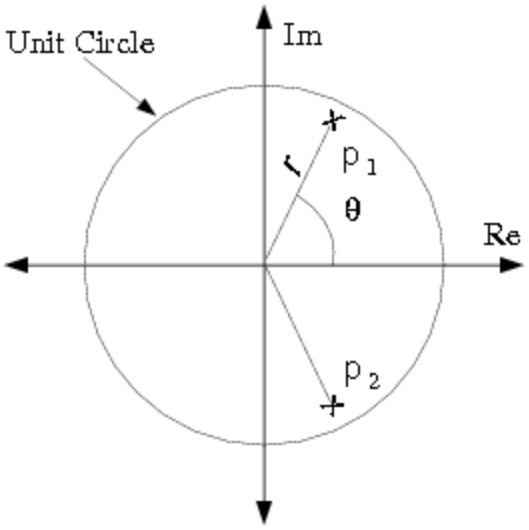
INLAB REPORT
For both the original audio signal and the filtered output, hand in the following:

- The time domain plot of 101 samples.
- The plot of the magnitude of the DTFT for 1001 samples.

Also hand in the code for the `FIRfilter` filtering function. Comment on how the frequency content of the signal changed after filtering. Is the filter we used a lowpass, highpass, bandpass, or a bandstop filter? Comment on how the filtering changed the quality of the audio signal.

Design of A Simple IIR Filter

Download the file pcm.mat for the following section.



Location of two poles for a simple IIR filter.

While zeros attenuate a filtered signal, poles amplify signals that are near their frequency. In this section, we will illustrate how poles can be used to separate a narrow-band signal from adjacent noise. Such filters are commonly used to separate modulated signals from background noise in applications such as radio-frequency demodulation.

Consider the following transfer function for a second order IIR filter with complex-conjugate poles:
Equation:

$$\begin{aligned} H_i(z) &= \frac{1-r}{\left(1-re^{j\theta}z^{-1}\right)\left(1-re^{-j\theta}z^{-1}\right)} \\ &= \frac{1-r}{1-2rcos(\theta)z^{-1}+r^2z^{-2}} \end{aligned}$$

[link] shows the locations of the two poles of this filter. The poles have the form

Equation:

$$p_1 = re^{j\theta} \quad p_2 = re^{-j\theta}$$

where r is the distance from the origin, and θ is the angle of p_1 relative to the positive real axis. From the theory of Z-transforms, we know that a causal filter is stable if and only if its poles are located within the unit circle. This implies that this filter is stable if and only if $|r| < 1$. However, we will see that by locating the poles close to the unit circle, the filter's bandwidth may be made extremely narrow around θ.

This two-pole system is an example of an IIR filter because its impulse response has infinite duration. Any filter with nontrivial poles (not at z = 0 or $\pm\infty$) is an IIR filter unless the poles are canceled by zeros.

Calculate the magnitude of the filter's frequency response $\left|H_i\left(e^{jw}\right)\right|$ on $|\omega| < \pi$ for $\theta = \pi/3$ and the following three values of r.

- **Equation:**

$$r = 0.99$$

- **Equation:**

$$r = 0.9$$

- **Equation:**

$$r = 0.7$$

Put all three plots on the same figure using the `subplot` command.

Note: Submit the difference equation, system diagram and the analytical expression of the impulse response for $H_i\left(z\right)$. Also submit the plot of the

magnitude of the frequency response for each value of r. Explain how the value of r affects this magnitude.

In the following experiment, we will use the filter $H_i\,(z)$ to separate a modulated sinusoid from background noise. To run the experiment, first download the file pcm.mat and load it into the Matlab workspace using the command `load pcm` . Play `pcm` using the `sound` command. Plot 101 samples of the signal for indices (100:200), and then compute the magnitude of the DTFT of 1001 samples of `pcm` using the time indices (100:1100). Plot the magnitude of the DTFT samples versus radial frequency for $|\omega| < \pi$. The two peaks in the spectrum correspond to the center frequency of the modulated signal. The low amplitude wideband content is the background noise. In this exercise, you will use the IIR filter described above to amplify the desired signal, relative to the background noise.

The pcm signal is modulated at 3146Hz and sampled at 8kHz. Use these values to calculate the value of θ for the filter $H_i\,(z)$. Remember from the sampling theorem that a radial frequency of 2π corresponds to the sampling frequency.

Write a Matlab function `IIRfilter(x)` that implements the filter $H_i\,(z)$. In this case, you need to use a **for** loop to implement the recursive difference equation. Use your calculated value of θ and $r = 0.995$. You can assume that y(n) is equal to 0 for negative values of n. Apply the new command `IIRfilter` to the signal `pcm` to separate the desired signal from the background noise, and listen to the filtered signal to hear the effects. Plot the filtered signal for indices (100:200), and then compute the DTFT of 1001 samples of the filtered signal using the time indices (100:1100). Plot the magnitude of this DTFT. In order to see the DTFT around $\omega = \theta$ more clearly, plot also the portion of this DTFT for the values of ω in the range $[\theta - 0.02, \theta + 0.02]$. Use your calculated value of θ.

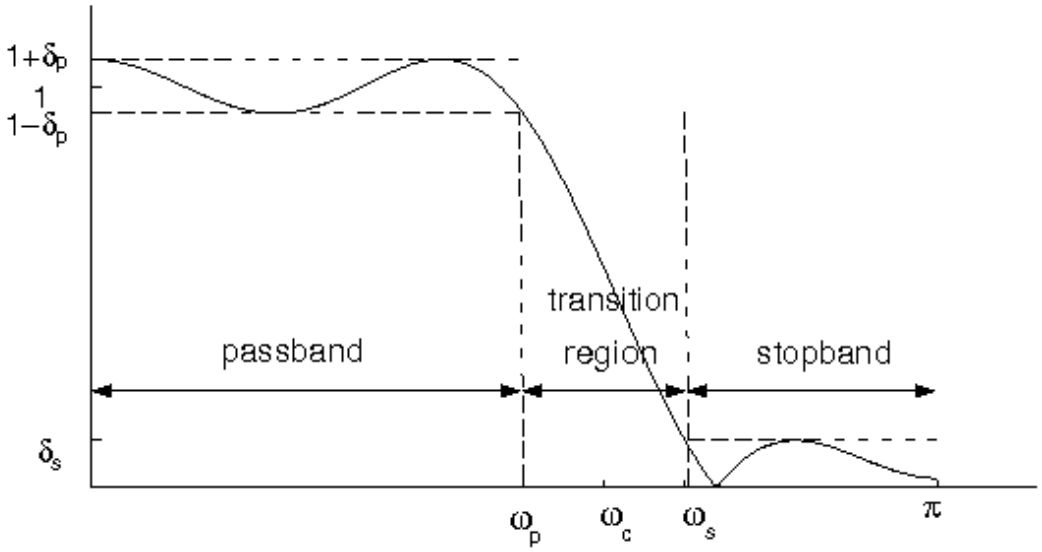
INLAB REPORT
For both the pcm signal and the filtered output, submit the following:

- The time domain plot of the signal for 101 points.
- The plot of the magnitude of the DTFT computed from 1001 samples of the signal.
- The plot of the magnitude of the DTFT for ω in the range $[\theta - 0.02, \theta + 0.02]$.

Also hand in the code for the `IIRfilter` filtering function. Comment on how the signal looks and sounds before and after filtering. How would you expect changes in r to change the filtered output? Would a value of $r = 0.9999999$ be effective for this application? Why might such a value for r be ill-advised? (Consider the spectrum of the desired signal around $\omega = \theta$.)

Lowpass Filter Design Parameters

Download the file nspeech2.mat for the following section.

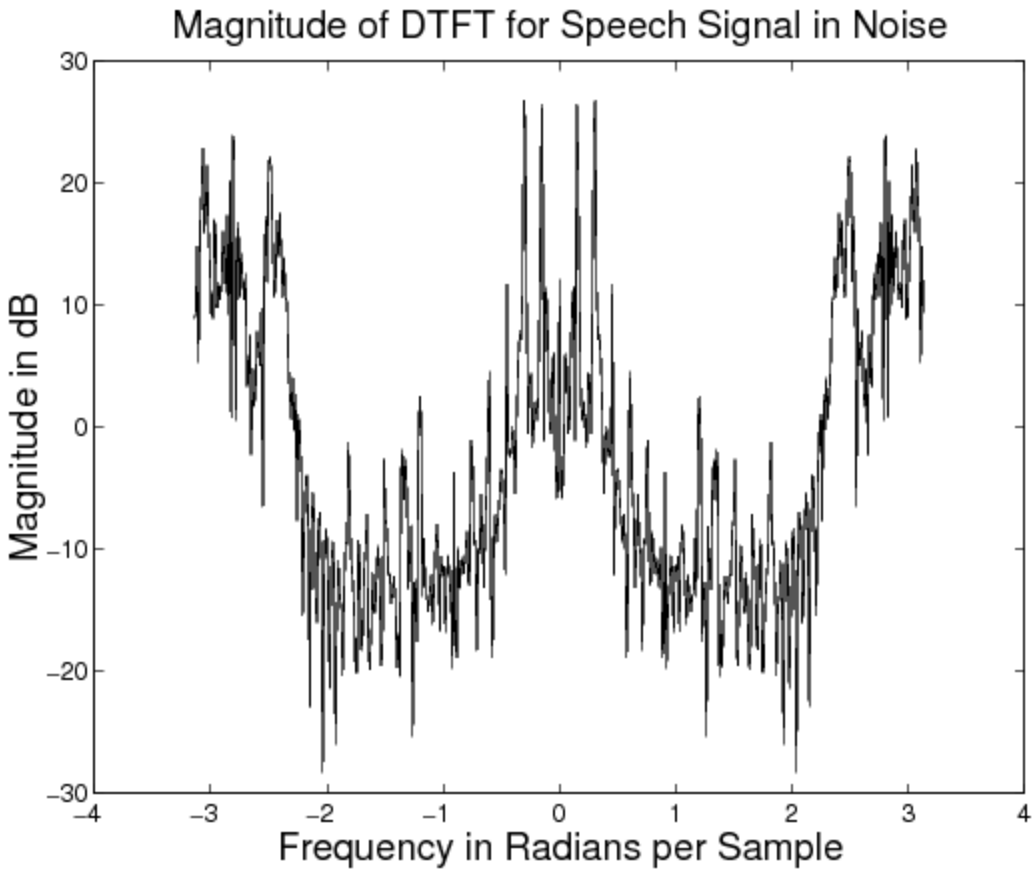


Tolerance specifications for the frequency response of a filter.

Oftentimes it is necessary to design a good approximation to an ideal lowpass, highpass or bandpass filter. [link] illustrates the typical characteristics of a real low-pass filter. The frequencies $|\omega| < \omega_p$ are known as the passband, and the frequencies $\omega_s < |\omega| \leq \pi$ are the stopband. For any real filter, $\omega_p < \omega_s$. The range of frequencies $\omega_p \leq \omega \leq \omega_s$ is known as the transition band. The magnitude of the filter response, $H\left(e^{j\omega}\right)$, is constrained in the passband and stopband by the following two equations
Equation:

$$\begin{aligned} \left|H\left(e^{j\omega}\right) - 1\right| &\leq \delta_p \quad \text{for } |\omega| < \omega_p \\ \left|H\left(e^{j\omega}\right)\right| &\leq \delta_s \quad \text{for } \omega_s < |\omega| \leq \pi \end{aligned}$$

where δ_p and δ_s are known as the passband and stopband ripple respectively. Most lowpass filter design techniques depend on the specification of these four parameters: ω_p, ω_s, δ_p, and δ_s.



DTFT of a section of noisy speech.

To illustrate the selection of these parameters consider the problem of filtering out background noise from a speech signal. [link] shows the magnitude of the DTFT over a window of such a signal, called nspeech2. Notice that there are two main components in nspeech2: one at the low frequencies and one at the high. The high frequency signal is noise, and it is band limited to $|\omega| > 2.2$. The low frequency signal is speech and it is band limited to $|\omega| < 1.8$. Download the file nspeech2.mat. and load it into the Matlab workspace. It contains the signal `nspeech2` from [link]. Play the `nspeech2` using the `sound` command and note the quality of the speech and background noise.

In the following sections, we will compute low-pass filters for separating the speech and noise using a number of different methods.

Filter Design Using Truncation

Ideally, a low-pass filter with cutoff frequency ω_c should have a frequency response of
Equation:

$$H_{ideal}\left(e^{jw}\right) = \begin{cases} 1 & |\omega| \leq \omega_c \\ 0 & \omega_c < |\omega| \leq \pi \end{cases}$$

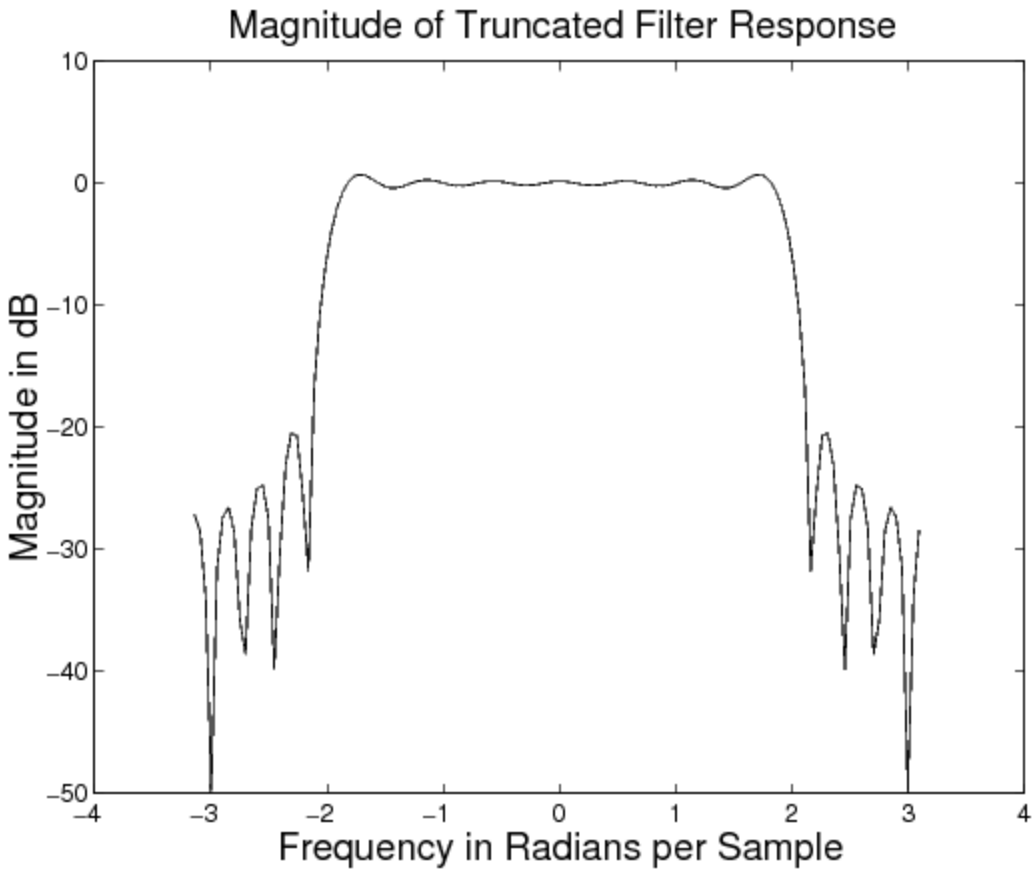
and a corresponding impulse response of
Equation:

$$h_{ideal}\left(n\right) = \frac{\omega_c}{\pi}\text{sinc}\left(\frac{\omega_c n}{\pi}\right) \quad \text{for} -\infty < n < \infty$$

However, no real filter can have this frequency response because $h_{ideal}\left(n\right)$ is infinite in duration.

One method for creating a realizable approximation to an ideal filter is to truncate this impulse response outside of $n \in [-M, M]$.
Equation:

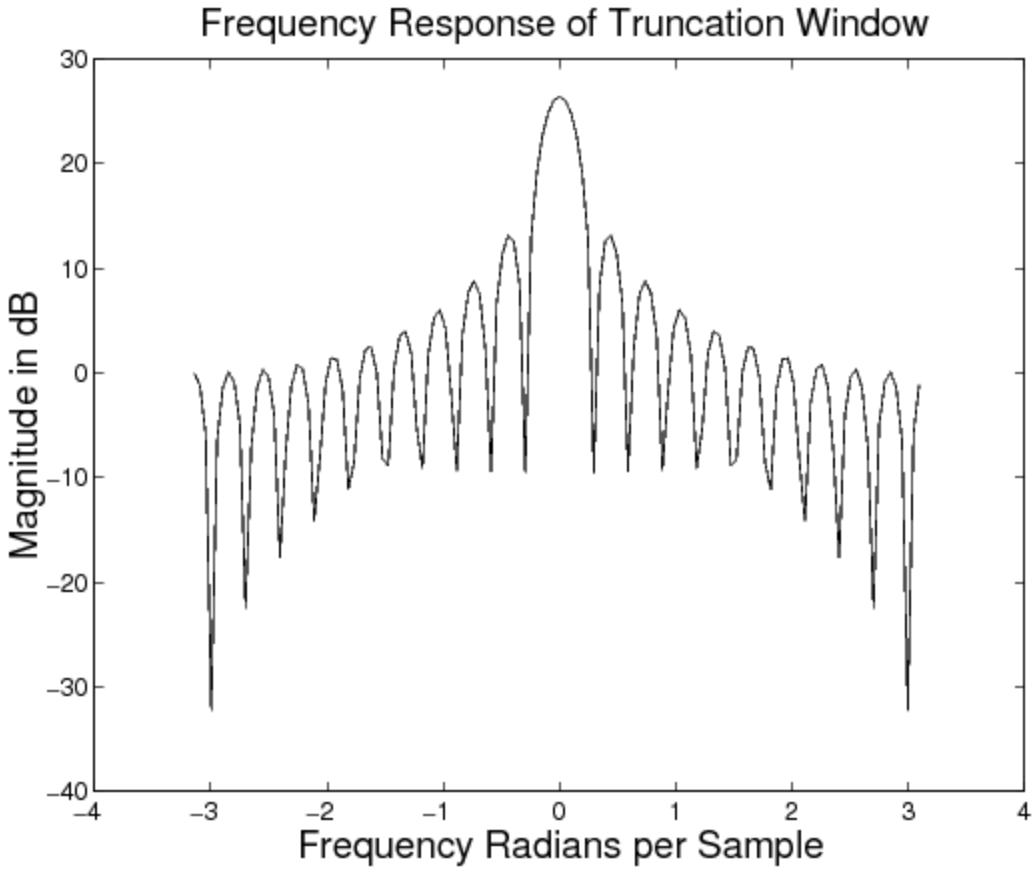
$$h_{trunc}\left(n\right) = \begin{cases} \frac{\omega_c}{\pi}\text{sinc}\left(\frac{\omega_c}{\pi}n\right) & n = -M, ..., 0, 1, ..., M \\ 0 & \text{otherwise} \end{cases}$$



Frequency response of low-pass filter designed using the truncation method.

[link] shows the magnitude response of the lowpass filter with cutoff frequency $\omega_c = 2.0$, with the impulse response truncated to $n \in [-10, 10]$. Notice the oscillatory behavior of the magnitude response near the cutoff frequency and the large amount of ripple in the stopband.

Due to the modulation property of the DTFT, the frequency response of the **truncated** filter is the result of convolving the magnitude response of the **ideal** filter (a rect) with the **DTFT of the truncating window**. The DTFT of the truncating window, shown in [link], is similar to a sinc function since it is the DTFT of a sampled rectangular window. Notice that this DTFT has very large sidelobes, which lead to large stopband ripple in the final filter.



DTFT of a rectangular window of size 21.

A truncated impulse response is of finite duration, yet the filter is still noncausal. In order to make the FIR filter causal, it must be shifted to the right by M units. For a filter of size $N = 2M + 1$ this shifted and truncated filter is given by

Equation:

$$h\left(n\right) = \begin{cases} \frac{\omega_c}{\pi} \operatorname{sinc}\left(\frac{\omega_c}{\pi}\left(n - \frac{N-1}{2}\right)\right) & n = 0, 1, ..., N-1 \\ 0 & \text{otherwise} \end{cases}.$$

This time shift of $(N-1)/2$ units to the right corresponds to multiplying the frequency response by $e^{-j\omega(N-1)/2}$. It does not affect the magnitude

response of the filter, but adds a factor of $-j\omega(N-1)/2$ to the phase response. Such a filter is called **linear phase** because the phase is a linear function of ω.

It is interesting to see that the filter formula of [link] is valid for N both even and odd. While both of these filters are linear phase, they have different characteristics in the time domain. When N is odd, then the value at $n = (N-1)/2$ is sampled at the peak of the sinc function, but when N is even, then the two values at $n = N/2$ and $n = (N/2) - 1$ straddle the peak.

To examine the effect of filter size on the frequency characteristics of the filter, write a Matlab function `LPFtrunc(N)` that computes the truncated and shifted impulse response of size N for a low pass filter with a cutoff frequency of $\omega_c = 2.0$. For each of the following filter sizes, plot the magnitude of the filter's DTFT in decibels. Hints: The magnitude of the response in decibels is given by $\left|H_{dB}\left(e^{j\omega}\right)\right| = 20\log_{10}\left|H\left(e^{j\omega}\right)\right|$. Note that the log command in Matlab computes the natural logarithm. Therefore, use the log10 command to compute decibels. To get an accurate representation of the DTFT make sure that you compute at least 512 sample points using the command `[X,w]=DTFT(filter_response,512)` .

- **Equation:**

$$N = 21$$

- **Equation:**

$$N = 101$$

Now download the noisy speech signal nspeech2.mat , and `load` it into the Matlab workspace. Apply the two filters with the above sizes to this signal. Since these are FIR filters, you can simply convolve them with the audio signal. Listen carefully to the unfiltered and filtered signals, and note the result. Can you hear a difference between the two filtered signals? In order to hear the filtered signals better, you may want to multiply each of them by 2 or 3 before using `sound`.

INLAB REPORT

- Submit the plots of the magnitude response for the two filters (not in decibels). On each of the plots, mark the passband, the transition band and the stopband.
- Submit the plots of the magnitude response in decibels for the two filters.
- Explain how the filter size effects the stopband ripple. Why does it have this effect?
- Comment on the quality of the filtered signals. Does the filter size have a noticeable effect on the audio quality?

Lab 5b - Digital Filter Design (part 2)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This is the second part of a two week laboratory in digital filter design. The first week of the laboratory covered some basic examples of FIR and IIR filters, and then introduced the concepts of filter design. In this week we will cover more systematic methods of designing both FIR and IIR filters.

Filter Design Using Standard Windows

Download DTFT.m for the following section.

We can generalize the idea of truncation by using different windowing functions to truncate an ideal filter's impulse response. Note that by simply truncating the ideal filter's impulse response, we are actually multiplying (or "windowing") the impulse response by a shifted $rect()$ function. This particular type of window is called a **rectangular** window. In general, the impulse reponse $h(n)$ of the designed filter is related to the impulse response $h_{ideal}\,(n)$ of the ideal filter by the relation
Equation:

$$h\,(n) = w\,(n) h_{ideal}\,(n),$$

where $w(n)$ is an N-point window. We assume that
Equation:

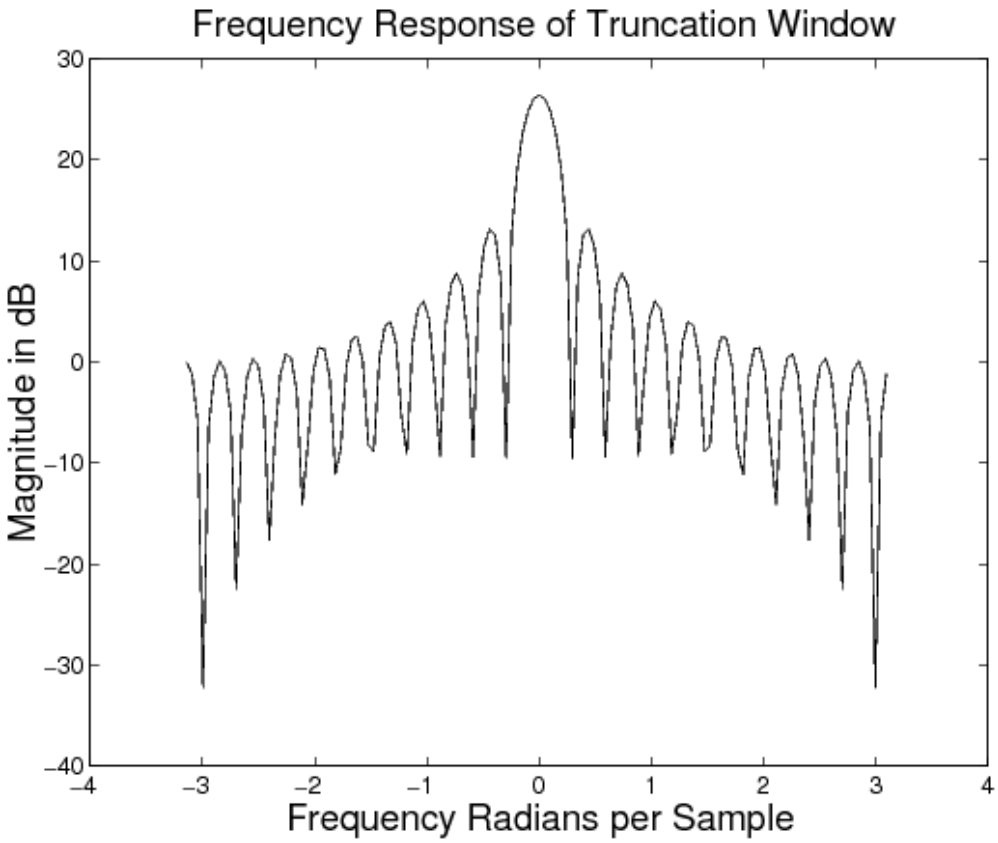
$$h_{ideal}\,(n) = \frac{\omega_c}{\pi} \mathrm{sinc}\left(\frac{\omega_c}{\pi} \left(n - \frac{N-1}{2} \right) \right),$$

where ω_c is the cutoff frequency and N is the desired window length.

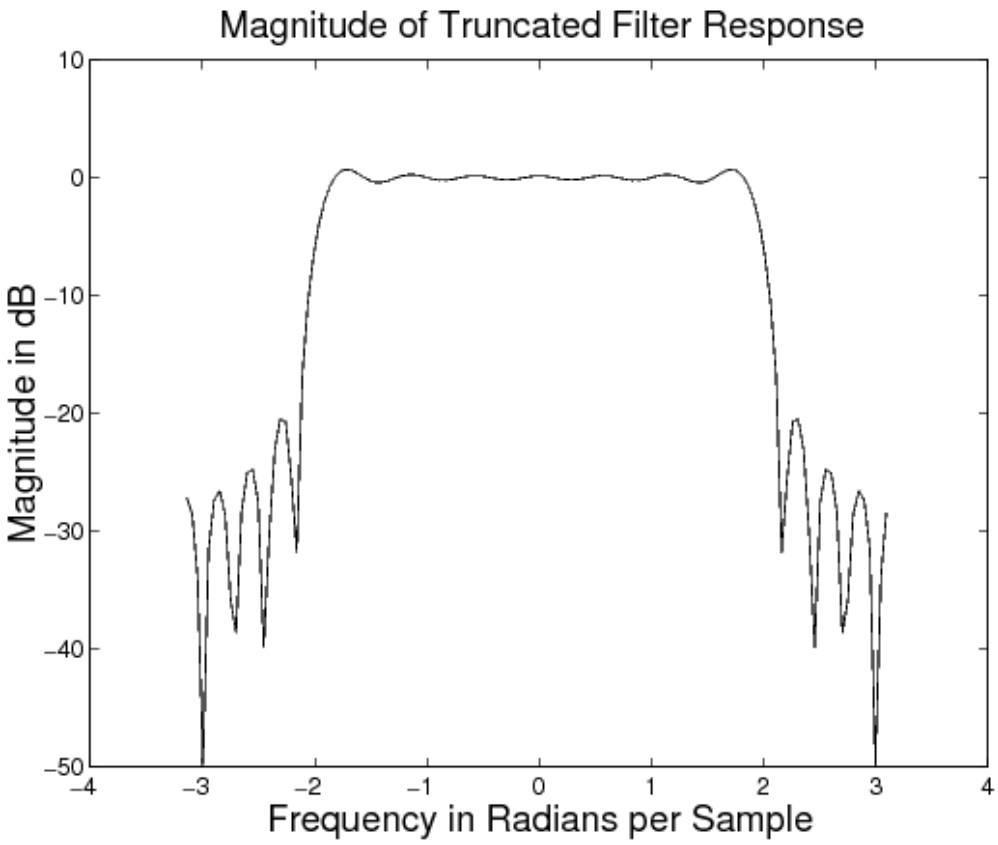
The rectangular window is defined as
Equation:

$$w\left(n\right) = \begin{cases} 1 & n = 0, 1, ..., N-1 \\ 0 & \text{otherwise} \end{cases}$$

The DTFT of $w(n)$ for $N = 21$ is shown in [link]. The rectangular window is usually not preferred because it leads to the large stopband and passband ripple as shown in [link].



DTFT of a rectangular window of length 21.



Frequency response of low-pass filter, designed using the truncation method.

More desirable frequency characteristics can be obtained by making a better selection for the window, $w(n)$. In fact, a variety of raised cosine windows are widely used for this purpose. Some popular windows are listed below.

1. Hanning window (as defined in Matlab, command `hann(N)`):
   Equation:

$$w\left(n\right)=\begin{cases}0.5-0.5\cos\frac{2\pi n}{N-1} & n=0,1,...,N-1\\ 0 & \text{otherwise}\end{cases}$$

2. Hamming window
   Equation:

$$w\left(n\right)=\begin{cases}0.54-0.46\cos\frac{2\pi n}{N-1} & n=0,1,...,N-1\\ 0 & \text{otherwise}\end{cases}$$

3. Blackman window
   Equation:

$$w\left(n\right)=\begin{cases}0.42-0.5\cos\frac{2\pi n}{N-1}+0.08\cos\frac{4\pi n}{N-1} & n=0,1,...,N-1\\ 0 & \text{otherwise}\end{cases}$$

In filter design using different truncation windows, there are two key frequency domain effects that are important to the design: the transition band **roll-off**, and the passband and stopband **ripple** (see [link] below). There are two corresponding parameters in the spectrum of each type of window that influence these filter parameters. The filter's roll-off is related to the **width** of center lobe of the window's magnitude spectrum. The **ripple** is influenced by the ratio of the mainlobe amplitude to the first sidelobe amplitude (or difference if using a dB scale). These two window spectrum parameters are **not** independent, and you should see a trend as you examine the spectra for different windows. The theoretical values for the **mainlobe width** and the **peak-to-sidelobe amplitude** are shown in [link].

Window (length N)	Mainlobe width	Peak-to-sidelobe amplitude (dB)
$Rectangular$	$4\pi/N$	$-13dB$
$Hanning$	$8\pi/N$	$-32dB$
$Hamming$	$8\pi/N$	$-43dB$
$Blackman$	$12\pi/N$	$-58dB$

Approximate spectral parameters of truncation windows. See reference [1].

Plot the rectangular, Hamming, Hanning, and Blackman window functions of length 21 on a single figure using the `subplot` command. You may use the Matlab commands `hamming`, `hann`, and `blackman`. Then compute and plot the DTFT magnitude of each of the four windows. Plot the magnitudes on a decibel scale (i.e., plot $20 \log_{10} \left| W\left(e^{j\omega}\right) \right|$). Download and use the function DTFT.m to compute the DTFT.

Note: Use at least 512 sample points in computing the DTFT by typing the command

```
DTFT(window,512)
```

. Type help DTFT for further information on this function.

Measure the null-to-null mainlobe width (in rad/sample) and the peak-to-sidelobe amplitude (in dB) from the logarithmic magnitude response plot for each window type. The Matlab command `zoom` is helpful for this. Make a table with these values **and** the theoretical ones.

Now use a Hamming window to design a lowpass filter $h(n)$ with a cutoff frequency of $\omega_c = 2.0$ and length 21. Note: You need to use [link] and [link] for this design. In the same figure, plot the filter's impulse response, and the magnitude of the filter's DTFT in decibels.

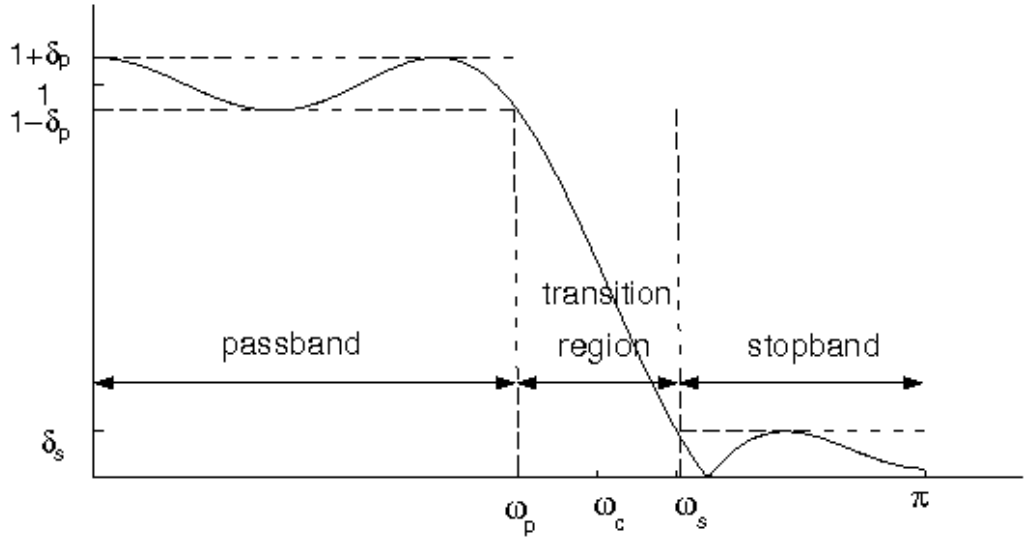
INLAB REPORT

1. Submit the figure containing the time domain plots of the four windows.
2. Submit the figure containing the DTFT (in decibels) of the four windows.
3. Submit the table of the measured and theoretical window spectrum parameters. Comment on how close the experimental results matched the ideal values. Also comment on the relation between the width of the mainlobe and the peak-to-sidelobe amplitude.

4. Submit the plots of your designed filter's impulse response and the magnitude of the filter's DTFT.

Filter Design Using the Kaiser Window

Download nspeech2.mat for the following section.



Tolerance specifications for the frequency response of a filter.

The standard windows of the "Filter Design Using Standard Windows" section are an improvement over simple truncation, but these windows still do not allow for arbitrary choices of transition bandwidth and ripple. In 1964, James Kaiser derived a family of near-optimal windows that can be used to design filters which meet or exceed any filter specification. The Kaiser window depends on two parameters: the window length N, and a parameter β which controls the shape of the window. Large values of β reduce the window sidelobes and therefore result in reduced passband and stopband ripple. The only restriction in the Kaiser filter design method is that the passband and stopband ripple must be equal in magnitude. Therefore, the Kaiser filter must be designed to meet the smaller of the two ripple constraints:

Equation:

$$\delta = \min\left\{\delta_p, \delta_s\right\}$$

The Kaiser window function of length N is given by
Equation:

$$w\left(n\right) = \begin{cases} \frac{I_0\left(\beta\frac{\sqrt{n(N-1-n)}}{N-1}\right)}{I_0(\beta)} & n = 0, 1, ..., N-1 \\ 0 & \text{otherwise} \end{cases}$$

where $I_0\left(\cdot\right)$ is the zero'th order modified Bessel function of the first kind, N is the length of the window, and β is the shape parameter.

Kaiser found that values of β and N could be chosen to meet any set of design parameters, $(\delta, \omega_p, \omega_s)$, by defining $A = -20\ \log_{10}\ \delta$ and using the following two equations:
Equation:

$$\beta = \begin{cases} 0.1102(A - 8.7) & A > 50 \\ 0.5842(A - 21)^{0.4} + 0.07886\left(A - 21\right) & 21 \leq A \leq 50 \\ 0.0 & A < 21 \end{cases}$$

Equation:

$$N = \left\lceil 1 + \frac{A - 8}{2.285(\omega_s - \omega_p)} \right\rceil$$

where $\lceil \cdot \rceil$ is the **ceiling** function, i.e. $\lceil x \rceil$ is the smallest integer which is greater than or equal to x.

To further investigate the Kaiser window, plot the Kaiser windows and their DTFT magnitudes (in dB) for $N = 21$ and the following values of β:

- **Equation:**

$$\beta = 0$$

- **Equation:**

$$\beta = 1$$

- **Equation:**

$$\beta = 5$$

For each case use at least 512 points in the plot of the DTFT.

Note: To create the Kaiser windows, use the Matlab command

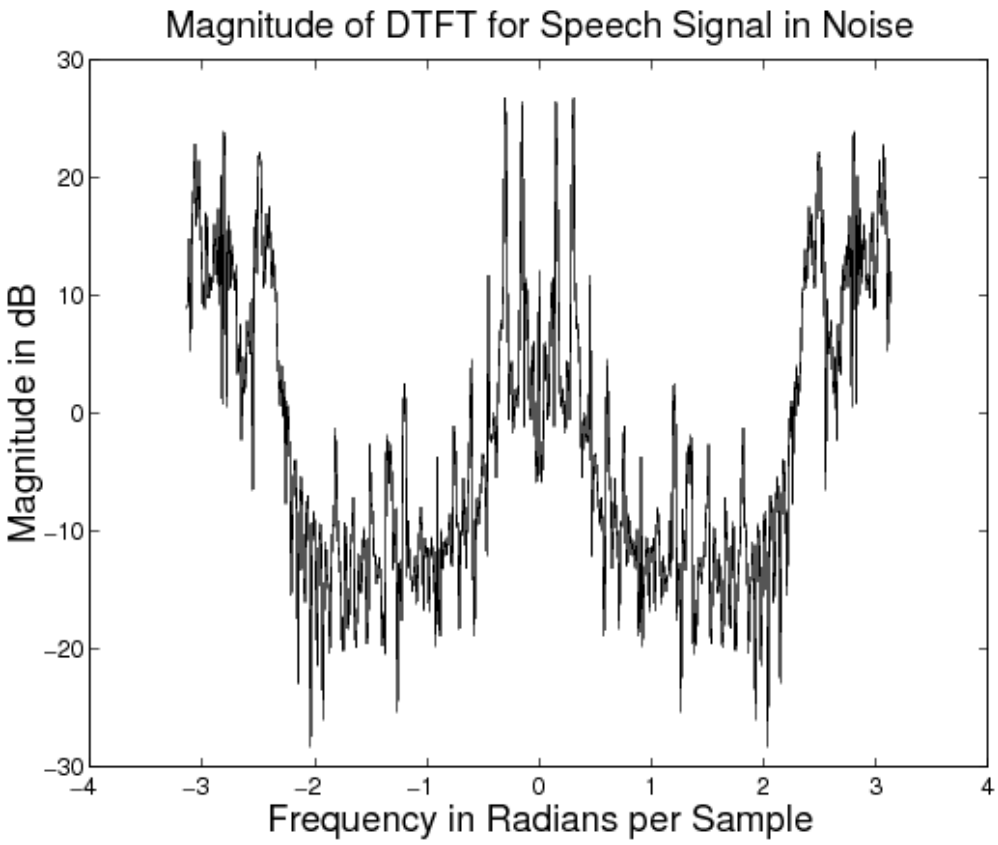
```
kaiser(N,beta)
```

command where N is the length of the filter and beta is the shape parameter β. To insure at least 512 points in the plot use the command

```
DTFT(window,512)
```

when computing the DTFT.

Note: Submit the plots of the 3 Kaiser windows and the magnitude of their DTFT's in decibels. Comment on how the value β affects the shape of the window and the sidelobes of the DTFT.



DTFT of a section of noisy speech.

Next use a Kaiser window to design a low pass filter, $h(n)$, to remove the noise from the signal in nspeech2.mat using equations [link] and [link]. To do this, use equations [link] and [link] to compute the values of β and N that will yield the following design specifications:

Equation:

$$\begin{aligned} \omega_p &= 1.8 \\ \omega_c &= 2.0 \\ \omega_s &= 2.2 \\ \delta_p &= 0.05 \\ \delta_s &= 0.005 \end{aligned}$$

The low pass filter designed with the Kaiser method will automatically have a cut-off frequency centered between ω_p and ω_s.
Equation:

$$\omega_c = \frac{\omega_p + \omega_s}{2}$$

Plot the magnitude of the DTFT of $h(n)$ for $|\omega| < \pi$. Create three plots in the same figure: one that shows the entire frequency response, and ones that zoom in on the passband and stopband ripple, respectively. Mark ω_p, ω_s, δ_p, and δ_s on these plots where appropriate. Note: Since the ripple is measured on a magnitude scale, DO NOT use a decibel scale on this set of plots.

From the Matlab prompt, compute the stopband and passband ripple (do not do this graphically). Record the stopband and passband ripple to three decimal places.

Note: To compute the passband ripple, find the value of the DTFT at frequencies corresponding to the passband using the command

```
H(abs(w)<=1.8)
```

where

```
H
```

is the DTFT of $h(n)$ and

```
w
```

is the corresponding vector of frequencies. Then use this vector to compute the passband ripple. Use a similar procedure for the stopband ripple.

Filter the noisy speech signal in nspeech2.mat using the filter you have designed. Then compute the DTFT of 400 samples of the filtered signal

starting at time $n = 20000$ (i.e. 20001:20400). Plot the magnitude of the DTFT samples in decibels versus frequency in radians for $|\omega| < \pi$. Compare this with the spectrum of the noisy speech signal shown in [link]. Play the noisy and filtered speech signals back using `sound` and listen to them carefully.

INLAB REPORT
Do the following:

1. Submit the values of β and N that you computed.
2. Submit the three plots of the filter's magnitude response. Make sure the plots are labeled.
3. Submit the values of the passband and stopband ripple. Does this filter meet the design specifications?
4. Submit the magnitude plot of the DTFT in dB for the filtered signal. Compare this plot to the plot of [link].
5. Comment on how the frequency content and the audio quality of the filtered signal have changed after filtering.

FIR Filter Design Using Parks-McClellan Algorithm

Click here for help on the `firpm` function for Parks-McClellan filter design. Download the data file nspeech2.mat for the following section.

Kaiser windows are versatile since they allow the design of arbitrary filters which meet specific design constraints. However, filters designed with Kaiser windows still have a number of disadvantages. For example,

- Kaiser filters are not guaranteed to be the minimum length filter which meets the design constraints.
- Kaiser filters do not allow passband and stopband ripple to be varied independently.

Minimizing filter length is important because in many applications the length of the filter determines the amount of computation. For example, an FIR filter of length N may be directly implemented in the time domain by evaluating the expression
Equation:

$$y\left(n\right) = \sum_{k=0}^{N-1} x\left(n-k\right) h\left(k\right) \ .$$

For each output value $y(n)$ this expression requires N multiplies and $N-1$ additions.

Oftentimes $h(n)$ is a symmetric filter so that $h(n) = h(N-1-n)$. If the filter $h(n)$ is symmetric and N is even, then [link] may be more efficiently computed as
Equation:

$$y\left(n\right) = \sum_{k=0}^{N/2-1} \left(x(n-k) + x(n-N+1+k)\right) h\left(k\right) \ .$$

This strategy reduces the computation to $N/2$ multiplies and $N-1$ adds for any value of N[footnote]. Note that the computational effort is linearly proportional to the length of the filter.
The advantages of using such symmetries varies considerably with the implementation and application. On many modern computing architectures the computational cost of adds and multiplies are similar, and the overhead of control loops may eliminate the advantages of reduced operations.

The Kaiser filters do not guarantee the minimum possible filter length. Since the filter has equal passband and stopband ripple, it will usually exceed design requirements in one of the two bands; this results in an unnecessarily long filter. A better design would allow the stopband and passband constraints to be specified separately.

In 1972, Parks and McClellan devised a methodology for designing symmetric filters that minimize filter length for a particular set of design constraints $\{\omega_p$, ω_s, δ_p, $\delta_s\}$. The resulting filters minimize the maximum error between the desired frequency response and the actual frequency response by spreading the approximation error uniformly over each band. The Parks and McClellan algorithm makes use of the Remez exchange algorithm and Chebyshev approximation theory. Such filters that exhibit **equiripple** behavior in both the passband and the stopband, and are sometimes called equiripple filters.

As with Kaiser filters, designing a filter with the Parks and McClellan algorithm is a two step process. First the length (i.e. order) of the filter must be computed based on the design constraints. Then the optimal filter for a specified length can be determined. As with Kaiser windows, the filter length computation is approximate so the resulting filter may exceed or violate the design constraints. This is generally not a problem since the filter can be redesigned for different lengths until the constraints are just met.

The Matlab command for computing the approximate filter length is

```
[n,fo,mo,w] = firpmord(f,m,ripple,2*pi)
```

where the inputs are:

- `f` - vector containing an even number of band edge frequencies. For a simple low pass filter, `f=[wp,ws]` , where `wp` and `ws` are the passband and stopband frequencies, respectively.
- `m` - vector containing the ideal filter magnitudes of the filter in each band. For a simple low pass filter `m=[1,0]` .
- `ripple` - vector containing the allowed ripple in each band. For a simple low pass filter `ripple=[delta_p,delta_s]` , where `delta_p` and `delta_s` are the passband and stopband ripples, respectively.
- `2*pi` - value, in radians, that corresponds to the sampling frequency.

The outputs of the command are `n = filter length - 1` , and the vectors `fo` , `mo` , and `w` which are intermediate filter parameters.

Once the filter length, `n` , is obtained, the Matlab command for designing a Parks-McClellan filter is `b = firpm(n,fo,mo,w)` . The inputs `n` , `fo` , `mo` , and `w` are the corresponding outputs of `firpmord`, and the output `b` is a vector of FIR filter coefficients such that
Equation:

$$H(z) = b(1) + b(2)z^{-1} + \cdots + b(n+1)z^{-n}$$

(What is the impulse response of this filter?)

For further information, read the help document on using Matlab to implement the Parks-McClellan algorithm.

Now design a symmetric FIR filter using `firpmord` and `firpm` in Matlab to meet the design specifications given in the "Filter Design Using the Kaiser Window" section. Compute the DTFT of the filter's response for at least 512 points, and use this result to compute the passband and stopband ripple of the filter that was designed. Adjust the filter length until the minimum order which meets the design constraints is found. Plot the magnitude of the DTFT in dB of the final filter design.

INLAB REPORT
Do the following:

1. Submit the final measured values of filter length, passband ripple, and stopband ripple. How accurate was the filter order computation using Matlab's `firpmord`? How does the length of this filter compare to the filter designed using a Kaiser window?
2. Submit the plot of the filter's DTFT. How does the frequency response of the Parks-McClellan filter compare to the filter designed using the Kaiser window? Comment on the shape of both the passband and stopband.

Use the filter you have designed to remove the noise from the signal `nspeech2.mat` . Play the noisy and filtered speech signals back using `sound` and listen to them carefully. Compute the DTFT of 400 samples of the filtered signal starting at time $n = 20,001$ (i.e. 20001:20400). Plot the magnitude of the DTFT in decibels versus frequency in radians for $|\omega| < \pi$. Compare this with the spectrum of the noisy speech signal shown in [link], and also with the magnitude of the DTFT of the Kaiser filtered signal.

Note: Submit the plot of the DTFT magnitude for the filtered signal. Comment on how the audio quality of the signal changes after filtering. Also comment on any differences in audio quality between the Parks-McClellan filtered speech and the Kaiser filtered speech.

Design of Discrete-Time IIR Filters Using Numerical Optimization

In this section, we consider the design of discrete-time IIR filters through the direct search of filter parameters that will minimize a specific design criterion. Such “brute force” approaches to filter design have become increasingly more popular due to the wide availability of high speed computers and robust numerical optimization methods.

Typically, numerical approaches to filter design have two parts. First, they design a **cost**, or **error** criterion. This criterion is a measure of the difference between the ideal filter response and the response of the computed or “approximate” filter. The goal is to find the approximate filter with the lowest cost (error). Mean square error is a popular cost criterion. The second part is to minimize the cost with respect to the filter parameters. We will perform the required numerical optimization with the `fminsearch` function in Matlab's **Optimization Toolbox**.

In order to formulate a cost criterion, we must first select a model for the discrete-time filter of interest. There are many ways of doing this, but we will use the coefficients of a rational transfer function to model (or parameterize) the set of second order IIR filters. In this case, the elements of the vector $\theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5]$ are the coefficients of the transfer function
Equation:

$$H_\theta(z) = \frac{\theta_1 + \theta_2 z^{-1} + \theta_3 z^{-2}}{1 + \theta_4 z^{-1} + \theta_5 z^{-2}} \ .$$

Using this parameterization, we may then define a function $\mathrm{Cost}(\theta)$ which is the “cost” of using the filter $H_\theta(z)$.

To illustrate this numerical optimization approach, we will design a digital filter that compensates for the roll-off due to the sample-and-hold process in an audio CD player. In lab 4, we saw that the sample-and-hold operation in a conventional D/A converter causes the reconstructed signal to be filtered by the function
Equation:

$$H_{sh}\left(e^{j\omega}\right) = \text{sinc}\left(\frac{\omega}{2\pi}\right) \quad \text{for } |\omega| < \pi$$

One method of reducing this distortion is to digitally “pre-filter” a signal with the inverse transfer function, $1/H_{sh}$. This filter $1/H_{sh}$ pre-distorts the audio signal so the reconstructed signal has the desired frequency response. We would like to approximate the filter $1/H_{sh}$ using the second order filter of [link].

For an audio CD player, the magnitude of the frequency response is generally considered to be more important than the phase. This is because we are not perceptually sensitive to phase distortion in sound. Therefore, we may choose a cost function which computes the total squared error between the magnitudes of the desired pre-filter response, $1/H_{sh}\left(e^{j\omega}\right)$, and the second order filter $H_\theta\left(e^{j\omega}\right)$:
Equation:

$$\text{Cost}\left(\theta\right) = \int_{-\pi}^{\pi}\left(\left|\frac{1}{H_{sh}\left(e^{j\omega}\right)}\right| - \left|H_\theta\left(e^{j\omega}\right)\right|\right)^2 d\omega$$

The θ parameters that minimize this cost function will be the parameters of our designed filter. A more complex approach might also account for the filter phase, but for simplicity we will only try to match the filter magnitudes.

After the filter is designed, we may compute the difference between the CD player's frequency response in dB and the ideal desired response in dB that the CD player should have:
Equation:

$$\text{Err}_{dB}\left(\omega\right) = 20\,\log_{10}\left(H_{sh}\left(e^{j\omega}\right)\left|H_{\theta^*}\left(e^{j\omega}\right)\right|\right)$$

where θ^* is the optimized value of θ and $H_{\theta^*}\left(e^{j\omega}\right)$ is the optimum second-order filter.

Do the following to perform this filter design:

- Write a Matlab function `prefilter(w,theta)` which computes the frequency response $H_\theta\left(e^{j\omega}\right)$ from equation [link] for the vector of input frequencies `w` and the parameter vector `theta`.
- Write a Matlab function `Cost(theta)` which computes the total squared error of equation [link]. Use a sampling interval $\Delta\omega = 0.01$ for the functions $H_\theta\left(e^{j\omega}\right)$ and $1/H_{sh}\left(e^{j\omega}\right)$.
- Use the command `fminsearch` from Matlab's **Optimization Toolbox** to compute the value of the parameter θ which minimizes `Cost(theta)`. The function `fminsearch` has the syntax `X = fminsearch('function_name',initial_value)` where function_name is the name of the function being minimized (`Cost`), `initial_value` is the starting value for the unknown parameter, and `X` is the minimizing parameter vector. Choose an initial value of $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = (1, 0, 0, 0, 0)$ so that $H_\theta\left(e^{j\omega}\right) = 1$.
- Use the `subplot` command to plot the following three functions on the interval $[-\pi, \pi]$.
    - The desired filter magnitude response $1/H_{sh}\left(e^{j\omega}\right)$.
    - The designed IIR filter magnitude response $\left|H_{\theta^*}\left(e^{j\omega}\right)\right|$.
    - The error in decibels, from equation [link].

INLAB REPORT
Do the following:

1. Submit the printouts of the code for the two Matlab functions `prefilter.m` and `Cost.m`.
2. Give an analytical expression for the optimized transfer function $H_{\theta^*}(z)$ with the coefficients that were computed.
3. Submit the three plots. On the error plot, mark the frequency ranges where the approximation error is high.

Lab 6a - Discrete Fourier Transform and FFT (part 1)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This is the first week of a two week laboratory that covers the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) methods. The first week will introduce the DFT and associated sampling and windowing effects, while the second week will continue the discussion of the DFT and introduce the FFT.

In previous laboratories, we have used the Discrete-Time Fourier Transform (DTFT) extensively for analyzing signals and linear time-invariant systems.
Equation:

$$(\text{DTFT}) \qquad X\left(e^{j\omega}\right) = \sum_{n=-\infty}^{\infty} x(n) e^{-j\omega n}$$

Equation:

$$(\text{inverse DTFT}) \qquad x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X\left(e^{j\omega}\right) e^{j\omega n} d\omega.$$

While the DTFT is very useful analytically, it usually cannot be exactly evaluated on a computer because [link] requires an infinite sum and [link] requires the evaluation of an integral.

The discrete Fourier transform (DFT) is a sampled version of the DTFT, hence it is better suited for numerical evaluation on computers.
Equation:

$$\text{(DFT)} \qquad X_N(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

Equation:

$$\text{(inverse DFT)} \qquad x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X_N(k) e^{j2\pi kn/N}$$

Here $X_N(k)$ is an N point DFT of $x(n)$. Note that $X_N(k)$ is a function of a discrete integer k, where k ranges from 0 to $N-1$.

In the following sections, we will study the derivation of the DFT from the DTFT, and several DFT implementations. The fastest and most important implementation is known as the fast Fourier transform (FFT). The FFT algorithm is one of the cornerstones of signal processing.

Deriving the DFT from the DTFT

Truncating the Time-domain Signal

The DTFT usually cannot be computed exactly because the sum in [link] is infinite. However, the DTFT may be approximately computed by truncating the sum to a finite window. Let $w(n)$ be a rectangular window of length N:
Equation:

$$w(n) = \begin{cases} 1 & 0 \le n \le N-1 \\ 0 & \text{else} \end{cases}.$$

Then we may define a truncated signal to be
Equation:

$$x_{\text{tr}}(n) = w(n) x(n) \ .$$

The DTFT of $x_{\text{tr}}(n)$ is given by:
Equation:

$$\begin{aligned} X_{\text{tr}}\left(e^{j\omega}\right) &= \sum_{n=-\infty}^{\infty} x_{\text{tr}}\left(n\right)e^{-j\omega n} \\ &= \sum_{n=0}^{N-1} x\left(n\right)e^{-j\omega n} \quad . \end{aligned}$$

We would like to compute $X\left(e^{j\omega}\right)$, but as with filter design, the truncation window distorts the desired frequency characteristics; $X\left(e^{j\omega}\right)$ and $X_{\text{tr}}\left(e^{j\omega}\right)$ are generally not equal. To understand the relation between these two DTFT's, we need to convolve in the frequency domain (as we did in designing filters with the truncation technique):
Equation:

$$X_{\text{tr}}\left(e^{j\omega}\right) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X\left(e^{j\sigma}\right)W\left(e^{j(\omega-\sigma)}\right)d\sigma$$

where $W\left(e^{j\omega}\right)$ is the DTFT of $w(n)$. [link] is the periodic convolution of $X\left(e^{j\omega}\right)$ and $W\left(e^{j\omega}\right)$. Hence the true DTFT, $X\left(e^{j\omega}\right)$, is smoothed via convolution with $W\left(e^{j\omega}\right)$ to produce the truncated DTFT, $X_{\text{tr}}\left(e^{j\omega}\right)$.

We can calculate $W\left(e^{j\omega}\right)$:
Equation:

$$\begin{aligned} W\left(e^{j\omega}\right) &= \sum_{n=-\infty}^{\infty} w\left(n\right)e^{-j\omega n} \\ &= \sum_{n=0}^{N-1} e^{-j\omega n} \\ &= \begin{cases} \frac{1-e^{-j\omega N}}{1-e^{-j\omega}}, & \text{for } \omega \neq 0, \pm 2\pi, ... \\ N, & \text{for } \omega = 0, \pm 2\pi, ... \end{cases} \end{aligned}$$

For $\omega \neq 0, \pm 2\pi, ...$, we have:

Equation:

$$\begin{aligned} W\left(e^{j\omega}\right) &= \frac{e^{-j\omega N/2}}{e^{-j\omega/2}} \frac{e^{j\omega N/2} - e^{-j\omega N/2}}{e^{j\omega/2} - e^{-j\omega/2}} \\ &= e^{-j\omega(N-1)/2} \frac{\sin\left(\omega N/2\right)}{\sin\left(\omega/2\right)} \ . \end{aligned}$$

Notice that the magnitude of this function is similar to $\text{sinc}(\omega N/2)$ except that it is periodic in ω with period 2π.

Frequency Sampling

[link] contains a summation over a finite number of terms. However, we can only evaluate [link] for a finite set of frequencies, ω. We must sample in the frequency domain to compute the DTFT on a computer. We can pick any set of frequency points at which to evaluate [link], but it is particularly useful to uniformly sample ω at N points, in the range $[0, 2\pi)$. If we substitute
Equation:

$$\omega = 2\pi k/N$$

for $k = 0, 1, ...(N-1)$ in [link], we find that
Equation:

$$\begin{aligned} X_{\text{tr}}\left(e^{j\omega}\right)\Big|_{\omega=\frac{2\pi k}{N}} &= \sum_{n=0}^{N-1} x\left(n\right) e^{-j\omega n}\Bigg|_{\omega=\frac{2\pi k}{N}} \\ &= \sum_{n=0}^{N-1} x\left(n\right) e^{-j2\pi kn/N} \\ &= X_N\left(k\right) \ . \end{aligned}$$

In short, the DFT values result from sampling the DTFT of the truncated signal.
Equation:

$$X_N(k) = X_{\text{tr}}\left(e^{j2\pi k/N}\right)$$

Windowing Effects

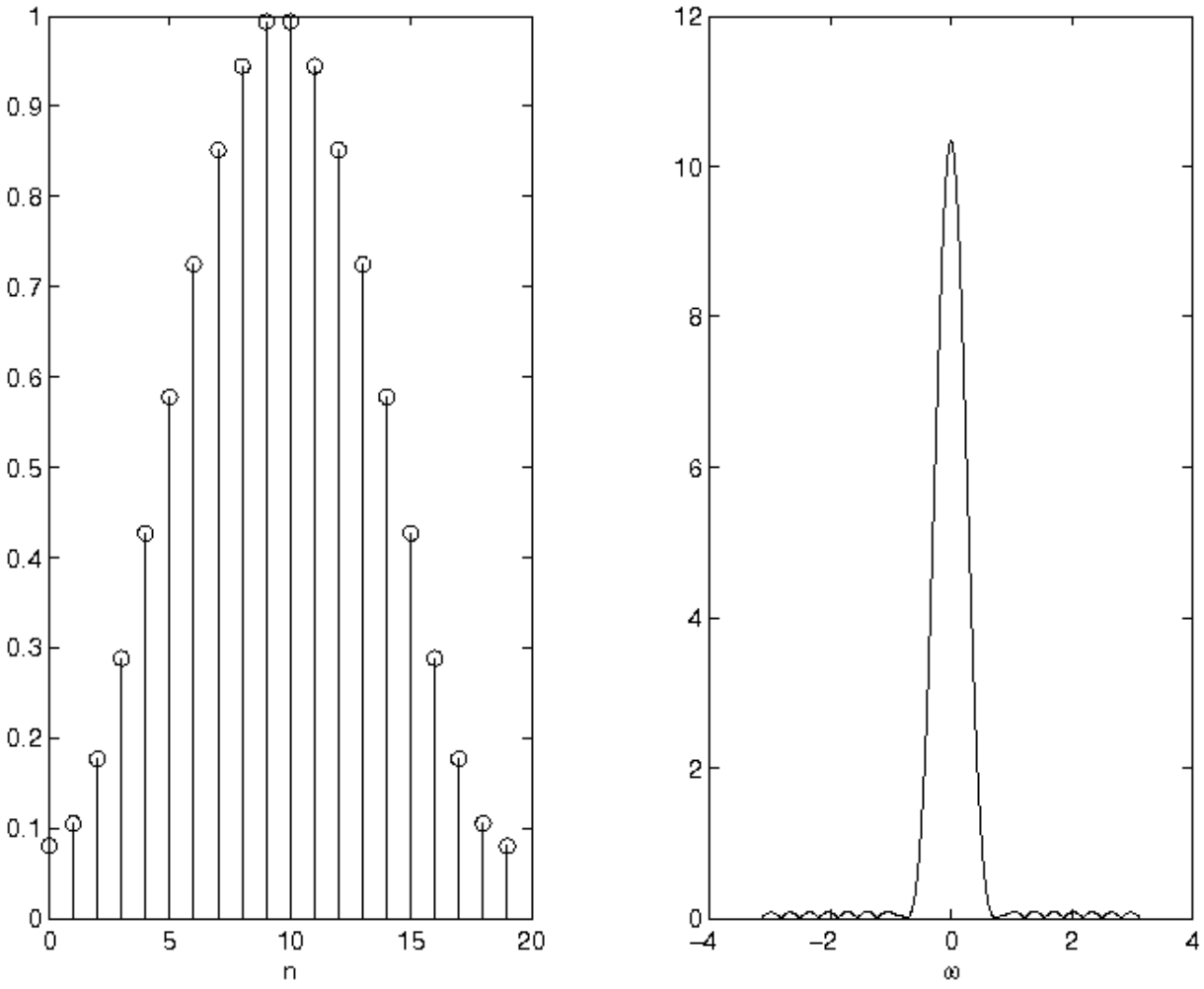
Download DTFT.m for the following section.

We will next investigate the effect of windowing when computing the DFT of the signal $x(n) = \cos\left(\frac{\pi}{4}n\right)$ truncated with a window of size $N = 20$.

1. In the same figure, plot the phase and magnitude of $W\left(e^{j\omega}\right)$, using equations [link] and [link].
2. Determine an expression for $X\left(e^{j\omega}\right)$ (the DTFT of the non-truncated signal).
3. Truncate the signal $x(n)$ using a window of size $N = 20$ and then use DTFT.m to compute $X_{\text{tr}}\left(e^{j\omega}\right)$. Make sure that the plot contains a least 512 points.

> **Note:** Use the command `[X,w] = DTFT(x,512)` .

4. Plot the magnitude of $X_{\text{tr}}\left(e^{j\omega}\right)$.



The plot of a Hamming window (left) and its DTFT (right).

INLAB REPORT

1. Submit the plot of the phase and magnitude of $W\left(e^{j\omega}\right)$.
2. Submit the analytical expression for $X\left(e^{j\omega}\right)$.
3. Submit the magnitude plot of $X_{\text{tr}}\left(e^{j\omega}\right)$.
4. Describe the difference between $\left|X_{\text{tr}}\left(e^{j\omega}\right)\right|$ and $\left|X\left(e^{j\omega}\right)\right|$. What is the reason for this difference?
5. Comment on the effects of using a different window for $w(n)$. For example, what would you expect your plots to look like if you had

used a Hamming window in place of the truncation (rectangular) window? (See [link] for a plot of a Hamming window of length 20 and its DTFT.)

The Discrete Fourier Transform

Computing the DFT

We will now develop our own DFT functions to help our understanding of how the DFT comes from the DTFT. Write your own Matlab function to implement the DFT of equation [link]. Use the syntax

```
X = DFTsum(x)
```

where x is an N point vector containing the values $x(0), \cdots, x(N-1)$ and X is the corresponding DFT. Your routine should implement the DFT exactly as specified by [link] using `for-loops` for n and k, and compute the exponentials as they appear. Note: In Matlab, "j" may be computed with the command `j=sqrt(-1)` . If you use $j = \sqrt{-1}$, remember not to use j as an index in your `for-loop`.

Test your routine `DFTsum` by computing $X_N\,(k)$ for each of the following cases:

1. $x(n) = \delta(n)$ for $N = 10$.
2. $x(n) = 1$ for $N = 10$.
3. $x\,(n) = e^{j2\pi n/10}$ for $N = 10$.
4. $x(n) =$cos $(2\pi n/10)$ for $N = 10$.

Plot the magnitude of each of the DFT's. In addition, derive simple closed-form analytical expressions for the DFT (not the DTFT!) of each signal.

INLAB REPORT

1. Submit a listing of your code for `DFTsum`.
2. Submit the magnitude plots.
3. Submit the corresponding analytical expressions.

Write a second Matlab function for computing the inverse DFT of [link]. Use the syntax

```
x = IDFTsum(X)
```

where `X` is the N point vector containing the DFT and `x` is the corresponding time-domain signal. Use `IDFTsum` to invert each of the DFT's computed in the previous problem. Plot the magnitudes of the inverted DFT's, and verify that those time-domain signals match the original ones. Use `abs(x)` to eliminate any imaginary parts which roundoff error may produce.

INLAB REPORT

1. Submit the listing of your code for `IDFTsum`.
2. Submit the four time-domain IDFT plots.

Matrix Representation of the DFT

The DFT of [link] can be implemented as a matrix-vector product. To see this, consider the equation
Equation:

$$X = \mathbf{A}x$$

where $\mathbf{A}$ is an $N \times N$ matrix, and both X and x are $N \times 1$ column vectors. This matrix product is equivalent to the summation
Equation:

$$X_k = \sum_{n=1}^{N} \mathbf{A}_{kn} x_n.$$

where $\mathbf{A}_{kn}$ is the matrix element in the k^{th} row and n^{th} column of A . By comparing [link] and [link] we see that for the DFT,

Equation:

$$\mathbf{A}_{kn} = e^{-j2\pi(k-1)(n-1)/N}$$

The -1's are in the exponent because Matlab indices start at 1, not 0. For this section, we need to:

- Write a Matlab function `A = DFTmatrix(N)` that returns the NxN DFT matrix A.

> **Note:** Remember that the symbol * is used for matrix multiplication in Matlab, and that $.'$ performs a simple transpose on a vector or matrix. An apostrophe without the period is a conjugate transpose.

- Use the matrix A to compute the DFT of the following signals. Confirm that the results are the same as in the previous section.

  1. $x(n) = \delta(n)$ for $N = 10$
  2. $x(n) = 1$ for $N = 10$
  3. $x\,(n) = e^{j2\pi n/N}$ for $N = 10$

INLAB REPORT

1. Print out the matrix $\mathbf{A}$ for $N = 5$.
2. Hand in the three magnitude plots of the DFT's.
3. How many multiplies are required to compute an N point DFT using the matrix method? (Consider a multiply as the multiplication of either complex or real numbers.)

As with the DFT, the inverse DFT may also be represented as a matrix-vector product.

Equation:

$$x = \mathbf{B}X$$

For this section,

1. Write an analytical expression for the elements of the inverse DFT matrix $\mathbf{B}$, using the form of [link].
2. Write a Matlab function `B = IDFTmatrix(N)` that returns the NxN inverse DFT matrix B.
3. Compute the matrices $\mathbf{A}$ and $\mathbf{B}$ for $N = 5$. Then compute the matrix product $\mathbf{C} = \mathbf{BA}$.

INLAB REPORT

1. Hand in your analytical expression for the elements of $\mathbf{B}$.
2. Print out the matrix $\mathbf{B}$ for $N = 5$.
3. Print out the elements of $\mathbf{C} = \mathbf{BA}$. What form does $\mathbf{C}$ have? Why does it have this form?

Computation Time Comparison

Click here for help on the `cputime` function.

Although the operations performed by `DFTsum` are mathematically identical to a matrix product, the computation times for these two DFT's in Matlab are quite different. (This is despite the fact that the computational complexity of two procedures is of the same order!) This exercise will underscore why you should try to avoid using **for loops** in Matlab, and wherever possible, try to formulate your computations using matrix/vector products.

To see this, do the following:

1. Compute the signal $x(n) = \cos(2\pi n/10)$ for $N = 512$.
2. Compute the matrix $\mathbf{A}$ for $N = 512$.
3. Compare the computation time of `X = DFTsum(x)` with a matrix implementation `X = A*x` by using the `cputime` function before and

after the program execution. Do not include the computation of `A` in your timing calculations.

Note: Report the

`cputime`

required for each of the two implementations. Which method is faster? Which method requires less storage?

Lab 6b - Discrete Fourier Transform and FFT (part 2)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This is the second week of a two week laboratory that covers the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT). The first week introduced the DFT and associated sampling and windowing effects. This laboratory will continue the discussion of the DFT and will introduce the FFT.

Continuation of DFT Analysis

This section continues the analysis of the DFT started in the previous week's laboratory.
Equation:

$$\begin{aligned} (\text{DFT}) \qquad X_N(k) &= \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \\ (\text{inverse DFT}) \qquad x(n) &= \frac{1}{N} \sum_{k=0}^{N-1} X_N(k) e^{j2\pi kn/N} \end{aligned}$$

Shifting the Frequency Range

In this section, we will illustrate a representation for the DFT of [link] that is a bit more intuitive. First create a Hamming window x of length $N = 20$, using the Matlab command `x = hamming(20)` . Then use your Matlab function `DFTsum` to compute the 20 point DFT of x. Plot the magnitude of the DFT, $|X_{20}(k)|$, versus the index k. Remember that the DFT index k starts at 0 not 1!

Note: Hand in the plot of the $|X_{20}(k)|$. Circle the regions of the plot corresponding to low frequency components.

Our plot of the DFT has two disadvantages. First, the DFT values are plotted against k rather then the frequency ω. Second, the arrangement of frequency samples in the DFT goes from 0 to 2π rather than from $-\pi$ to π, as is conventional with the DTFT. In order to plot the DFT values similar to a conventional DTFT plot, we must compute the vector of frequencies in radians per sample, and then "rotate" the plot to produce the more familiar range, $-\pi$ to π.

Let's first consider the vector w of frequencies in radians per sample. Each element of w should be the frequency of the corresponding DFT sample $X(k)$, which can be computed by

Equation:

$$\omega = 2\pi k/N \qquad k \in [0, \cdots, N-1] \ .$$

However, the frequencies should also lie in the range from $-\pi$ to π. Therefore, if $\omega \geq \pi$, then it should be set to $\omega - 2\pi$. An easy way of making this change in Matlab 5.1 is `w(w>=pi) = w(w>=pi)-2*pi` .

The resulting vectors X and w are correct, but out of order. To reorder them, we must swap the first and second halves of the vectors. Fortunately, Matlab provides a function specifically for this purpose, called **fftshift**.

Write a Matlab function to compute samples of the DTFT and their corresponding frequencies in the range $-\pi$ to π. Use the syntax

```
[X,w] = DTFTsamples(x)
```

where `x` is an N point vector, `X` is the length N vector of DTFT samples, and `w` is the length N vector of corresponding radial frequencies. Your function `DTFTsamples` should call your function `DFTsum` and use the Matlab function `fftshift`.

Use your function `DTFTsamples` to compute DTFT samples of the Hamming window of length $N = 20$. Plot the magnitude of these DTFT samples versus

frequency in rad/sample.

Note: Hand in the code for your function

`DTFTsamples`

. Also hand in the plot of the magnitude of the DTFT samples.

Zero Padding

The spacing between samples of the DTFT is determined by the number of points in the DFT. This can lead to surprising results when the number of samples is too small. In order to illustrate this effect, consider the finite-duration signal
Equation:

$$x\left(n\right)=\begin{cases}\sin\left(0.1\pi n\right) & 0\leq n\leq 49\\ 0 & \text{otherwise}\end{cases}$$

In the following, you will compute the DTFT samples of $x(n)$ using both $N=50$ and $N=200$ point DFT's. Notice that when $N=200$, most of the samples of $x(n)$ will be zeros because $x(n)=0$ for $n\geq 50$. This technique is known as “zero padding”, and may be used to produce a finer sampling of the DTFT.

For $N=50$ and $N=200$, do the following:

1. Compute the vector x containing the values $x(0),\cdots,x(N-1)$.
2. Compute the samples of $X(k)$ using your function `DTFTsamples`.
3. Plot the magnitude of the DTFT samples versus frequency in rad/sample.

Note: Submit your two plots of the DTFT samples for $N=50$ and $N=200$. Which plot looks more like the true DTFT? Explain why the plots look so different.

The Fast Fourier Transform Algorithm

We have seen in the preceding sections that the DFT is a very computationally intensive operation. In 1965, Cooley and Tukey ([link]) published an algorithm that could be used to compute the DFT much more efficiently. Various forms of their algorithm, which came to be known as the fast Fourier transform (FFT), had actually been developed much earlier by other mathematicians (even dating back to Gauss). It was their paper, however, which stimulated a revolution in the field of signal processing.

It is important to keep in mind at the outset that the FFT is **not** a new transform. It is simply a very efficient way to compute an existing transform, namely the DFT. As we saw, a straight forward implementation of the DFT can be computationally expensive because the number of multiplies grows as the square of the input length (i.e. N^2 for an N point DFT). The FFT reduces this computation using two simple but important concepts. The first concept, known as divide-and-conquer, splits the problem into two smaller problems. The second concept, known as recursion, applies this divide-and-conquer method repeatedly until the problem is solved.

Consider the defining equation for the DFT and assume that N is even, so that $N/2$ is an integer:
Equation:

$$X\left(k\right)=\sum_{n=0}^{N-1}x\left(n\right)e^{-j2\pi kn/N}.$$

Here we have dropped the subscript of N in the notation for $X(k)$. We will also use the notation
Equation:

$$X\left(k\right)=\mathrm{DFT}_N[x(n)]$$

to denote the N point DFT of the signal $x(n)$.

Suppose we break the sum in [link] into two sums, one containing all the terms for which n is even, and one containing all the terms for which n is odd:
Equation:

$$X(k) = \sum_{\substack{n=0 \\ n \text{ even}}}^{N-1} x(n) e^{-j2\pi kn/N} + \sum_{\substack{n=0 \\ n \text{ odd}}}^{N-1} x(n) e^{-j2\pi kn/N}.$$

We can eliminate the conditions "n even" and "n odd" in [link] by making a change of variable in each sum. In the first sum, we replace n by $2m$. Then as we sum m from 0 to $N/2-1$, $n = 2m$ will take on all even integer values between 0 and $N-2$. Similarly, in the second sum, we replace n by $2m+1$. Then as we sum m from 0 to $N/2-1$, $n = 2m+1$ will take on all odd integer values between 0 and $N-1$. Thus, we can write
Equation:

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) e^{-j2\pi k2m/N} + \sum_{m=0}^{N/2-1} x(2m+1) e^{-j2\pi k(2m+1)/N}.$$

Next we rearrange the exponent of the complex exponential in the first sum, and split and rearrange the exponent in the second sum to yield
Equation:

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) e^{-j2\pi km/(N/2)} + e^{-j2\pi k/N} \sum_{m=0}^{N/2-1} x(2m+1) e^{-j2\pi km/(N/2)}.$$

Now compare the first sum in [link] with the definition for the DFT given by [link]. They have exactly the same form if we replace N everywhere in [link] by $N/2$. Thus the first sum in [link] is an $N/2$ point DFT of the even-numbered data points in the original sequence. Similarly, the second sum in [link] is an $N/2$ point DFT of the odd-numbered data points in the original sequence. To obtain the N point DFT of the complete sequence, we multiply the DFT of the odd-numbered data points by the complex exponential factor $e^{-j2\pi k/N}$, and then simply sum the two $N/2$ point DFTs.

To summarize, we will rewrite [link] according to this interpretation. First, we define two new $N/2$ point data sequences $x_0(n)$ and $x_1(n)$, which contain the even and odd-numbered data points from the original N point sequence, respectively:
Equation:

$$\begin{aligned} x_0(n) &= x(2n) \\ x_1(n) &= x(2n+1), \end{aligned}$$

where $n = 0, \ldots, N/2 - 1$. This separation of even and odd points is called **decimation in time**. The N point DFT of $x(n)$ is then given by
Equation:

$$X(k) = X_0(k) + e^{-j2\pi k/N} X_1(k) \text{ for } k = 0, \ldots, N-1.$$

where $X_0(k)$ and $X_1(k)$ are the $N/2$ point DFT's of the even and odd points.
Equation:

$$\begin{aligned} X_0(k) &= \text{DFT}_{N/2}[x_0(n)] \\ X_1(k) &= \text{DFT}_{N/2}[x_1(n)] \end{aligned}$$

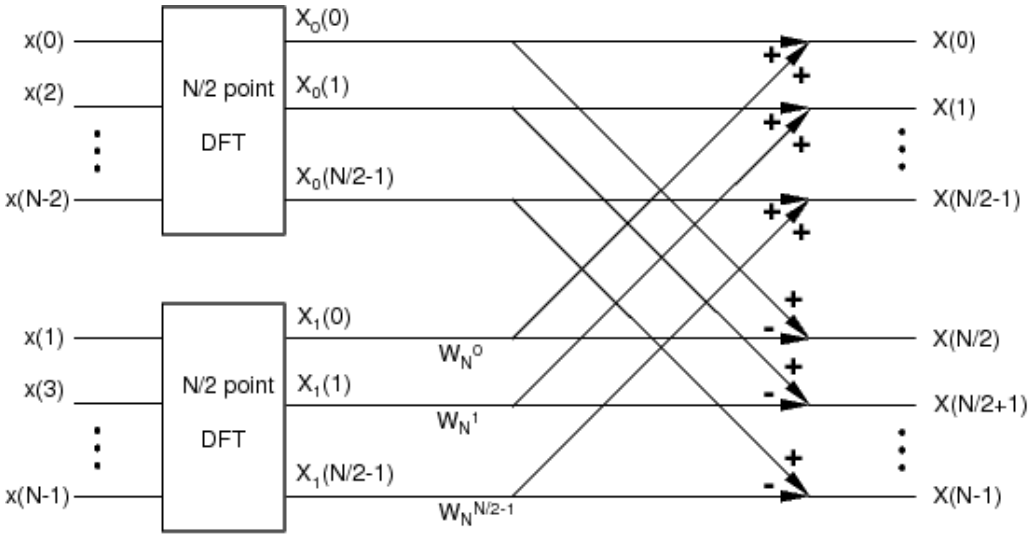
While [link] requires less computation than the original N point DFT, it can still be further simplified. First, note that each $N/2$ point DFT is periodic with period $N/2$. This means that we need to only compute $X_0(k)$ and $X_1(k)$ for $N/2$ values of k rather than the N values shown in [link]. Furthermore, the complex exponential factor $e^{-j2\pi k/N}$ has the property that
Equation:

$$-e^{-j2\pi\frac{k}{N}} = e^{-j2\pi\frac{k+N/2}{N}} \quad .$$

These two facts may be combined to yield a simpler expression for the N point DFT:
Equation:

$$\left.\begin{aligned} X(k) &= X_0(k) + W_N^k X_1(k) \\ X(k+N/2) &= X_0(k) - W_N^k X_1(k) \end{aligned}\right\} \text{ for } k = 0, \ldots, N/2-1$$

where the complex constants defined by $W_N^k = e^{-j2\pi k/N}$ are commonly known as the **twiddle factors**.



Divide and conquer DFT of equation (13). The N-point DFT is computed using the two $N/2$-point DFT's $X_0^{(N/2)}(k)$ and $X_1^{(N/2)}(k)$.

[link] shows a graphical interpretation of [link] which we will refer to as the "divide-and-conquer DFT". We start on the left side with the data separated into even and odd subsets. We perform an $N/2$ point DFT on each subset, and then multiply the output of the odd DFT by the required twiddle factors. The first half of the output is computed by adding the two branches, while the second half is formed by subtraction. This type of flow diagram is conventionally used to describe a fast Fourier transform algorithm.

Implementation of Divide-and-Conquer DFT

In this section, you will implement the DFT transformation using [link] and the illustration in [link]. Write a Matlab function with the syntax

```
X = dcDFT(x)
```

where `x` is a vector of even length N, and `X` is its DFT. Your function `dcDFT` should do the following:

1. Separate the samples of x into even and odd points.

> **Note:** The Matlab command `x0 = x(1:2:N)` can be used to obtain the "even" points.

2. Use your function `DFTsum` to compute the two $N/2$ point DFT's.
3. Multiply by the twiddle factors $W_N^k = e^{-j2\pi k/N}$.
4. Combine the two DFT's to form X.

Test your function `dcDFT` by using it to compute the DFT's of the following signals.

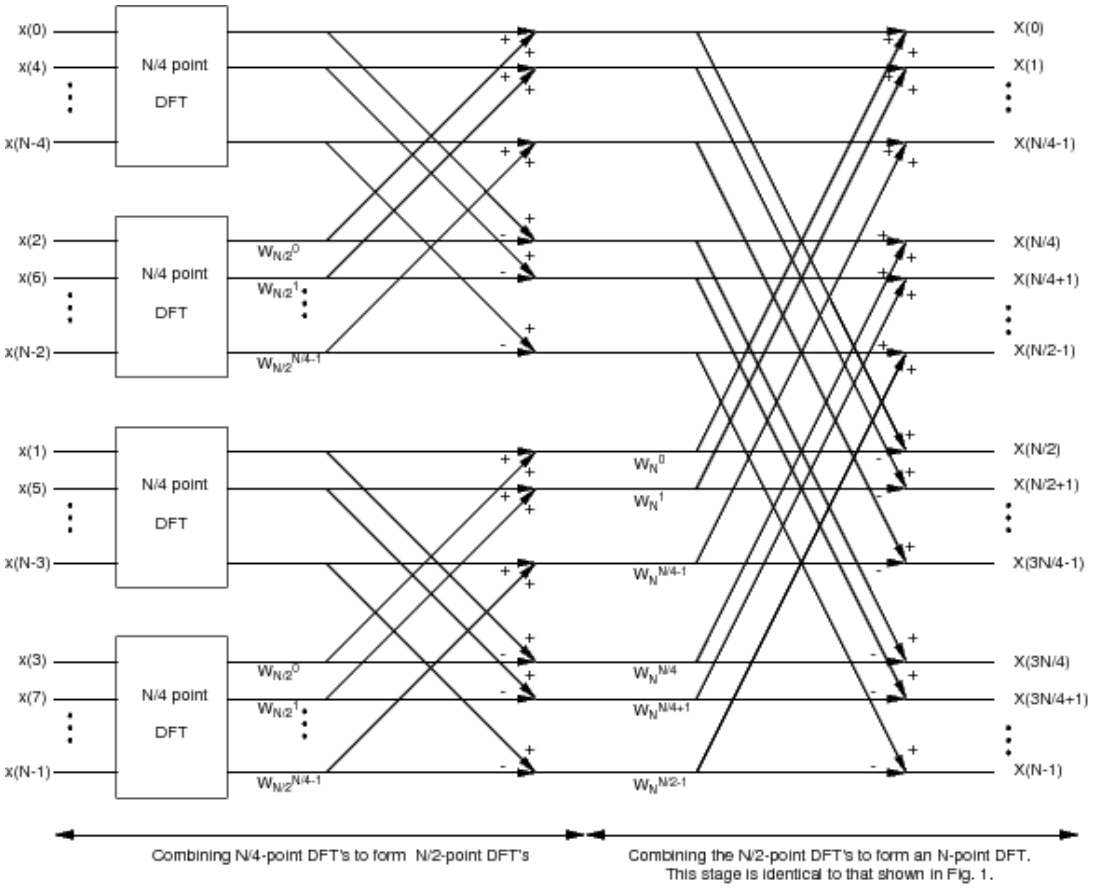
1. $x(n) = \delta(n)$ for $N = 10$
2. $x(n) = 1$ for $N = 10$
3. $x\,(n) = e^{j2\pi n/N}$ for $N = 10$

INLAB REPORT

1. Submit the code for your function `dcDFT`.
2. Determine the number of multiplies that are required in this approach to computing an N point DFT. (Consider a multiply to be one multiplication of real or complex numbers.)

> **Note:** Refer to the diagram of [link], and remember to consider the $N/2$ point DFTs.

Recursive Divide and Conquer



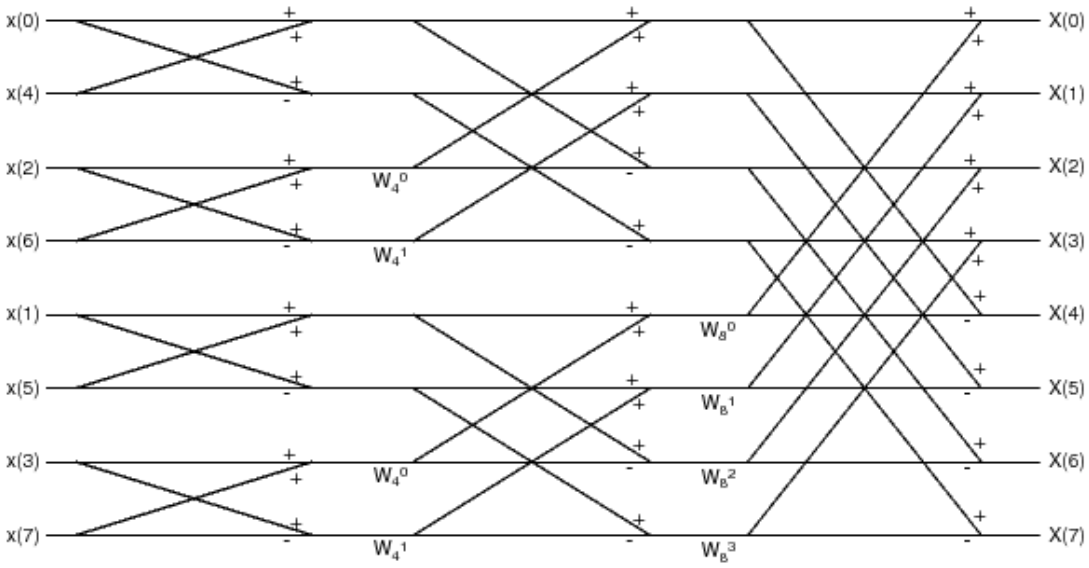
Recursion of the decimation-in-time process. Now each $N/2$-point is calculated by combining two $N/4$-point DFT's.

The second basic concept underlying the FFT algorithm is that of recursion. Suppose $N/2$ is also even. Then we may apply the same decimation-in-time idea to the computation of each of the $N/2$ point DFT's in [link]. This yields the process depicted in [link]. We now have two stages of twiddle factors instead of one.

How many times can we repeat the process of decimating the input sequence? Suppose N is a power of 2, i.e. $N = 2^p$ for some integer p. We can then repeatedly decimate the sequence until each subsequence contains only two points. It is easily seen from [link] that the 2 point DFT is a simple sum and difference of values.

Equation:

$$
\begin{aligned}
X(0) &= x(0) + x(1) \\
X(1) &= x(0) - x(1)
\end{aligned}
$$



8-Point FFT.

[link] shows the flow diagram that results for an 8 point DFT when we decimate 3 times. Note that there are 3 stages of twiddle factors (in the first stage, the twiddle factors simplify to “1”). This is the flow diagram for the complete decimation-in-time 8 point FFT algorithm. How many multiplies are required to compute it?

Write three Matlab functions to compute the 2, 4, and 8 point FFT's using the syntax

```
X = FFT2(x)
```

```
X = FFT4(x)
```

```
X = FFT8(x)
```

The function `FFT2` should directly compute the 2-point DFT using [link], but the functions `FFT4` and `FFT8` should compute their respective FFT's using the divide and conquer strategy. This means that `FFT8` should call `FFT4`, and `FFT4` should call `FFT2`.

Test your function `FFT8` by using it to compute the DFT's of the following signals. Compare these results to the previous ones.

1. $x(n) = \delta(n)$ for $N = 8$
2. $x(n) = 1 \quad$ for $N = 8$
3. $x\,(n) = e^{j2\pi n/8}$ for $N = 8$

INLAB REPORT

1. Submit the code for your functions `FFT2`, `FFT4` and `FFT8`.
2. List the output of `FFT8` for the case $x(n) = 1$ for $N = 8$.
3. Calculate the total number of multiplies **by twiddle factors** required for your 8 point FFT. (A multiply is a multiplication by a real or complex number.)
4. Determine a formula for the number of multiplies required for an $N = 2^p$ point FFT. Leave the expression in terms of N and p. How does this compare to the number of multiplies required for direct implementation when p=10?

If you wrote the **FFT4** and **FFT8** functions properly, they should have almost the exact same form. The only difference between them is the length of the input signal, and the function called to compute the (N/2)-pt DFTs. Obviously, it's redundant to write a separate function for each specific length DFT when they each have the same form. The preferred method is to write a **recursive** function, which means that the function calls itself within the body. It is imperative that a recursive function has a condition for exiting without calling itself, otherwise it would never terminate.

Write a recursive function `X=fft_stage(x)` that performs one stage of the FFT algorithm for a power-of-2 length signal. An outline of the function is as follows:

1. Determine the length of the input signal.
2. If N=2, then the function should just compute the 2-pt DFT as in [link], and then return.

3. If N>2, then the function should perform the FFT steps described previously (i.e. decimate, compute (N/2)-pt DFTs, re-combine), calling **fft_stage** to compute the (N/2)-pt DFTs.

Note that the body of this function should look very similar to previous functions written in this lab. Test **fft_stage** on the three 8-pt signals given above, and verify that it returns the same results as **FFT8**.

Note: Submit the code for your **fft_stage** function.

Lab 7a - Discrete-Time Random Processes (part 1)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

Many of the phenomena that occur in nature have uncertainty and are best characterized statistically as random processes. For example, the thermal noise in electronic circuits, radar detection, and games of chance are best modeled and analyzed in terms of statistical averages.

This lab will cover some basic methods of analyzing random processes. "Random Variables" reviews some basic definitions and terminology associated with random variables, observations, and estimation. "Estimating the Cumulative Distribution Function" investigates a common estimate of the cumulative distribution function. "Generating Samples from a Given Distribution" discusses the problem of transforming a random variable so that it has a given distribution, and lastly, "Estimating the Probability Density Function" illustrates how the **histogram** may be used to estimate the probability density function.

Note that this lab assumes an introductory background in probability theory. Some review is provided, but it is unfeasible to develop the theory in detail. A secondary reference such as [link] is strongly encouraged.

Random Variables

The following section contains an abbreviated review of some of the basic definitions associated with random variables. Then we will discuss the concept of an **observation** of a random event, and introduce the notion of an **estimator**.

Basic Definitions

A **random variable** is a function that maps a set of possible outcomes of a random experiment into a set of real numbers. The probability of an event can then be interpreted as the probability that the random variable will take on a value in a corresponding subset of the real line. This allows a fully numerical approach to modeling probabilistic behavior.

A very important function used to characterize a random variable is the **cumulative distribution function (CDF)**, defined as
Equation:

$$F_X(x) = P(X \leq x) \qquad x \in (-\infty, \infty) \ .$$

Here, X is the random variable, and $F_X(x)$ is the probability that X will take on a value in the interval $(-\infty, x]$. It is important to realize that x is simply a dummy variable for the function $F_X(x)$, and is therefore not random at all.

The derivative of the cumulative distribution function, if it exists, is known as the **probability density** function, denoted as $f_X(x)$. By the fundamental theorem of calculus, the probability density has the following property:
Equation:

$$\begin{aligned} \int_{t_0}^{t_1} f_X(x)dx &= F_X(t_1) - F_X(t_0) \\ &= P(t_0 < X \leq t_1) \ . \end{aligned}$$

Since the probability that X lies in the interval $(-\infty, \infty)$ equals one, the entire area under the density function must also equal one.

Expectations are fundamental quantities associated with random variables. The expected value of some function of X, call it $g(X)$, is defined by the following.
Equation:

$$
\begin{aligned}
E[g(X)] &= \int_{-\infty}^{\infty} g(x) f_X(x) dx \qquad (\text{for X continuous}) \\
E[g(X)] &= \sum_{x=-\infty}^{\infty} g(x) P(X = x) \qquad (\text{for X discrete})
\end{aligned}
$$

Note that expected value of $g(X)$ is a deterministic number. Note also that due to the properties of integration, expectation is a linear operator.

The two most common expectations are the mean μ_X and variance σ_X^2 defined by
Equation:

$$
\mu_X = E[X] = \int_{-\infty}^{\infty} x f_X(x) dx
$$

Equation:

$$
\sigma_X^2 = E\left[(X - \mu_X)^2\right] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) dx \ .
$$

A very important type of random variable is the **Gaussian** or **normal** random variable. A Gaussian random variable has a density function of the following form:
Equation:

$$
f_X(x) = \frac{1}{\sqrt{2\pi}\sigma_X} \exp\left(-\frac{1}{2\sigma_X^2}(x - \mu_X)^2\right) \ .
$$

Note that a Gaussian random variable is completely characterized by its mean and variance. This is not necessarily the case for other types of distributions. Sometimes, the notation $X \sim N(\mu, \sigma^2)$ is used to identify X as being Gaussian with mean μ and variance σ^2.

Samples of a Random Variable

Suppose some random experiment may be characterized by a random variable X whose distribution is unknown. For example, suppose we are measuring a deterministic quantity v, but our measurement is subject to a random measurement error ε. We can then characterize the observed value, X, as a random variable, $X = v + \varepsilon$.

If the distribution of X does not change over time, we may gain further insight into X by making several independent observations $\{X_1, X_2, \cdots, X_N\}$. These observations X_i, also known as **samples**, will be independent random variables and have the same distribution $F_X(x)$. In this situation, the X_i's are referred to as **i.i.d.**, for **independent** and **identically distributed**. We also sometimes refer to $\{X_1, X_2, \cdots, X_N\}$ collectively as a sample, or observation, of size N.

Suppose we want to use our observation $\{X_1, X_2, \cdots, X_N\}$ to estimate the mean and variance of X. Two estimators which should already be familiar to you are the **sample mean** and **sample variance** defined by
Equation:

$$\widehat{\mu}_X = \frac{1}{N}\sum_{i=1}^{N} X_i$$

Equation:

$$\widehat{\sigma}_X^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(X_i - \widehat{\mu}_X\right)^2 \ .$$

It is important to realize that these sample estimates are functions of random variables, and are therefore themselves random variables. Therefore we can also talk about the statistical properties of the estimators. For example, we can compute the mean and variance of the sample mean $\widehat{\mu}_X$.
Equation:

$$E\left[\widehat{\mu}_X\right] = E\left[\frac{1}{N}\sum_{i=1}^{N} X_i\right] = \frac{1}{N}\sum_{i=1}^{N} E[X_i] = \mu_X$$

Equation:

$$\begin{aligned} Var\left[\widehat{\mu}_X\right] &= Var\left[\frac{1}{N}\sum_{i=1}^{N} X_i\right] = \frac{1}{N^2} Var\left[\sum_{i=1}^{N} X_i\right] \\ &= \frac{1}{N^2}\sum_{i=1}^{N} Var[X_i] = \frac{\sigma_X^2}{N} \end{aligned}$$

In both [link] and [link] we have used the i.i.d. assumption. We can also show that $E\left[\widehat{\sigma}_X^2\right] = \sigma_X^2$.

An estimate $\widehat{a}$ for some parameter a which has the property $E\left[\widehat{a}\right] = a$ is said to be an **unbiased** estimate. An estimator such that $Var\left[\widehat{a}\right] \to 0$ as $N \to \infty$ is said to be **consistent**. These two properties are highly desirable because they imply that if a large number of samples are used the estimate will be close to the true parameter.

Suppose X is a Gaussian random variable with mean 0 and variance 1. Use the Matlab function `random` or `randn` to generate 1000 samples of X, denoted as X_1, X_2, ..., X_{1000}. See the online help for the random function. Plot them using the Matlab function `plot`. We will assume our generated samples are i.i.d.

Write Matlab functions to compute the sample mean and sample variance of [link] and [link] without using the predefined `mean` and `var` functions. Use these functions to compute the sample mean and sample variance of the samples you just generated.

INLAB REPORT

1. Submit the plot of samples of X.

2. Submit the sample mean and the sample variance that you calculated. Why are they not equal to the true mean and true variance?

Linear Transformation of a Random Variable

A linear transformation of a random variable X has the following form
Equation:

$$Y = aX + b$$

where a and b are real numbers, and $a \neq 0$. A very important property of linear transformations is that they are **distribution-preserving**, meaning that Y will be random variable with a distribution of the same form as X. For example, in [link], if X is Gaussian then Y will also be Gaussian, but not necessarily with the same mean and variance.

Using the linearity property of expectation, find the mean μ_Y and variance σ_Y^2 of Y in terms of a, b, μ_X, and σ_X^2. Show your derivation in detail.

Note: First find the mean, then substitute the result when finding the variance.

Consider a linear transformation of a Gaussian random variable X with mean 0 and variance 1. Calculate the constants a and b which make the mean and the variance of Y 3 and 9, respectively. Using [link], find the probability density function for Y.

Generate 1000 samples of X, and then calculate 1000 samples of Y by applying the linear transformation in [link], using the a and b that you just determined. Plot the resulting samples of Y, and use your functions to calculate the sample mean and sample variance of the samples of Y.

INLAB REPORT

1. Submit your derivation of the mean and variance of Y.
2. Submit the transformation you used, and the probability density function for Y.
3. Submit the plot of samples of Y and the Matlab code used to generate Y. Include the calculated sample mean and sample variance for Y.

Estimating the Cumulative Distribution Function

Suppose we want to model some phenomenon as a random variable X with distribution $F_X(x)$. How can we assess whether or not this is an accurate model? One method would be to make many observations and estimate the distribution function based on the observed values. If the distribution estimate is "close" to our proposed model $F_X(x)$, we have evidence that our model is a good characterization of the phenomenon. This section will introduce a common estimate of the cumulative distribution function.

Given a set of i.i.d. random variables $\{X_1, X_2, \ldots, X_N\}$ with CDF $F_X(x)$, the **empirical** cumulative distribution function $\widehat{F}_X(x)$ is defined as the following.
Equation:

$$\begin{aligned} \widehat{F}_X(x) &= \frac{1}{N}\sum_{i=1}^{N} I_{\{X_i \leq x\}} \\ I_{\{X_i \leq x\}} &= \begin{cases} 1, & \text{if } X_i \leq x \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

In words, $\widehat{F}_X(x)$ is the fraction of the X_i's which are less than or equal to x.

To get insight into the estimate $\widehat{F}_X(x)$, let's compute its mean and variance. To do so, it is easiest to first define N_x as the number of X_i's which are less than or equal to x.

Equation:

$$N_x = \sum_{i=1}^{N} I_{\{X_i \leq x\}} = N\widehat{F}_X(x)$$

Notice that $P(X_i \leq x) = F_X(x)$, so
Equation:

$$\begin{aligned} P\big(I_{\{X_i \leq x\}} = 1\big) &= F_X(x) \\ P\big(I_{\{X_i \leq x\}} = 0\big) &= 1 - F_X(x) \end{aligned}$$

Now we can compute the mean of $\widehat{F}_X(x)$ as follows,
Equation:

$$\begin{aligned} E\Big[\widehat{F}_X(x)\Big] &= \frac{1}{N} E[N_x] \\ &= \frac{1}{N} \sum_{i=1}^{N} E\big[I_{\{X_i \leq x\}}\big] \\ &= \frac{1}{N} N E\big[I_{\{X_i \leq x\}}\big] \\ &= 0 \cdot P\big(I_{\{X_i \leq x\}} = 0\big) + 1 \cdot P\big(I_{\{X_i \leq x\}} = 1\big) \\ &= F_X(x) \ . \end{aligned}$$

This shows that $\widehat{F}_X(x)$ is an unbiased estimate of $F_X(x)$. By a similar approach, we can show that
Equation:

$$Var\Big[\widehat{F}_X(x)\Big] = \frac{1}{N} F_X(x)\left(1 - F_X(x)\right) \ .$$

Therefore the empirical CDF $\widehat{F}_X(x)$ is both an unbiased and consistent estimate of the true CDF.

Exercise

Write a function `F=empcdf(X,t)` to compute the empirical CDF $\widehat{F}_X(t)$ from the sample vector X at the points specified in the vector t.

> **Note:** The expression `sum(X<=s)` will return the number of elements in the vector X which are less than or equal to s.

To test your function, generate a sample of `Uniform[0,1]` random variables using the function `X=rand(1,N)`. Plot two CDF estimates: one using a sample size $N = 20$, and one using $N = 200$. Plot these functions in the range `t=[-1:0.001:2]` , and on each plot superimpose the true distribution for a `Uniform[0,1]` random variable.

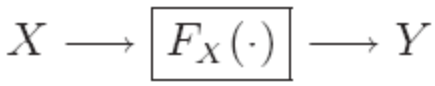
> **Note:**Hand in your
>
> `empcdf`
>
> function and the two plots.

Generating Samples from a Given Distribution

It is oftentimes necessary to generate samples from a particular distribution. For example, we might want to run simulations to test how an algorithm

performs on noisy inputs. In this section we will address the problem of generating random numbers from a given distribution $F_X(x)$.

Suppose we have a continuous random variable X with distribution $F_X(x)$, and we form the new random variable $Y = F_X(X)$. In other words Y is a function of X, and the particular function is the CDF of the random variable X.



Applying the transformation $F_X(\cdot)$ to X

.

How is Y distributed? First notice that $F_X(\cdot)$ is a probability, so that Y can only take values in the interval $[0, 1]$.
Equation:

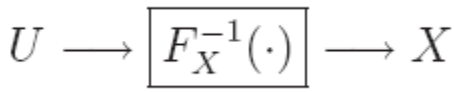
$$P(Y \leq y) = \begin{cases} 0, & \text{for } y < 0 \\ 1, & \text{for } y > 1 \end{cases}$$

Since $F_X(x)$ is a monotonically increasing function of x, the event $\{Y \leq y\}$ is equivalent to $\{X \leq x\}$ if we define $y = F_X(x)$. This implies that for $0 \leq y \leq 1$,
Equation:

$$
\begin{aligned}
F_Y(y) &= P(Y \leq y) \\
&= P(F_X(X) \leq F_X(x)) \\
&= P(X \leq x) && (\text{monotonicity}) \\
&= F_X(x) \\
&= y \ .
\end{aligned}
$$

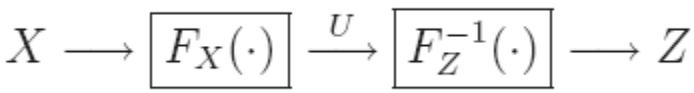
Therefore Y is uniformly distributed on the interval `[0,1]`.

Conversely, if $F_X(\cdot)$ is a one-to-one function, we may use the inverse transformation $F_X^{-1}(U)$ to transform a Uniform[0,1] random variable U to a random variable with distribution $F_X(\cdot)$.



Transforming a uniform random variable to one with distribution $F_X(\cdot)$.

Note that combining these results allows us to transform any continuous random variable $X \sim F_X(x)$ to any other continuous random variable $Z \sim F_Z(z)$, provided that $F_Z(\cdot)$ is a one-to-one function.



Transforming a random variable with

distribution $F_X(\cdot)$ to one with distibution $F_Z(\cdot)$.

Exercise

Your task is to use i.i.d. Uniform[0,1] random variables to generate a set of i.i.d. exponentially distributed random variables with CDF
Equation:

$$F_X(x) = \left(1 - e^{-3x}\right) u(x) \ .$$

Derive the required transformation.

Generate the `Uniform[0,1]` random variables using the function `rand(1,N)`. Use your `empcdf` function to plot two CDF estimates for the exponentially distributed random variables: one using a sample size $N = 20$, and one using $N = 200$. Plot these functions in the range `x=[-1:0.001:2]` , and on each plot superimpose the true exponential distribution of [link].

INLAB REPORT

- Hand in the derivation of the required transformation, and your Matlab code.
- Hand in the two plots.

Estimating the Probability Density Function

The statistical properties of a random variable are completely described by its probability density function (assuming it exists, of course). Therefore, it is oftentimes useful to estimate the PDF, given an observation of a random variable. For example, similar to the empirical CDF, probability density

estimates may be used to test a proposed model. They may also be used in non-parametric classification problems, where we need to classify data as belonging to a particular group but without any knowledge of the true underlying class distributions.

Notice that we cannot form a density estimate by simply differentiating the empirical CDF, since this function contains discontinuities at the sample locations X_i. Rather, we need to estimate the probability that a random variable will fall within a particular interval of the real axis. In this section, we will describe a common method known as the **histogram**.

The Histogram

Our goal is to estimate an arbitrary probability density function, $f_X(x)$, within a finite region of the x-axis. We will do this by partitioning the region into L equally spaced subintervals, or "bins", and forming an approximation for $f_X(x)$ within each bin. Let our region of support start at the value x_0, and end at x_L. Our L subintervals of this region will be $[x_0, x_1]$, $(x_1, x_2]$, ..., $(x_{L-1}, x_L]$. To simplify our notation we will define $bin(k)$ to represent the interval $(x_{k-1}, x_k]$, $k = 1, 2, \cdots, L$, and define the quantity Δ to be the length of each subinterval.
Equation:

$$\begin{aligned} bin(k) &= (x_{k-1}, x_k] \quad k = 1, 2, \cdots, L \\ \Delta &= \frac{x_L - x_0}{L} \end{aligned}$$

We will also define $\widetilde{f}(k)$ to be the probability that X falls into $bin(k)$.
Equation:

$$\begin{aligned} \widetilde{f}(k) &= P(X \in bin(k)) \\ &= \int_{x_{k-1}}^{x_k} f_X(x) dx \end{aligned}$$

Equation:

$$\approx \quad f_X(x)\Delta \quad \text{for } x \in bin(k)$$

The approximation in [link] only holds for an appropriately small bin width Δ.

Next we introduce the concept of a **histogram** of a collection of i.i.d. random variables $\{X_1, X_2, \cdots, X_N\}$. Let us start by defining a function that will indicate whether or not the random variable X_n falls within $bin(k)$.

Equation:

$$I_n(k) = \begin{cases} 1, & \text{if } X_n \in bin(k) \\ 0, & \text{if } X_n \notin bin(k) \end{cases}$$

The **histogram** of X_n at $bin(k)$, denoted as $H(k)$, is simply the number of random variables that fall within $bin(k)$. This can be written as

Equation:

$$H(k) = \sum_{n=1}^{N} I_n(k) \ .$$

We can show that the **normalized** histogram, $H(k)/N$, is an unbiased estimate of the probability of X falling in $bin(k)$. Let us compute the expected value of the normalized histogram.

Equation:

$$\begin{aligned} E\left[\frac{H(k)}{N}\right] &= \frac{1}{N}\sum_{n=1}^{N} E\left[I_n\left(k\right)\right] \\ &= \frac{1}{N}\sum_{n=1}^{N}\left\{1\cdot P\left(X_n \in bin\left(k\right)\right) + 0\cdot P\left(X_n \notin bin\left(k\right)\right)\right\} \\ &= \widetilde{f}\left(k\right) \end{aligned}$$

The last equality results from the definition of $\widetilde{f}\left(k\right)$, and from the assumption that the X_n's have the same distribution. A similar argument may be used to show that the variance of $H(k)$ is given by
Equation:

$$Var\left[\frac{H(k)}{N}\right] = \frac{1}{N}\widetilde{f}\left(k\right)\left(1-\widetilde{f}\left(k\right)\right) \ .$$

Therefore, as N grows large, the bin probabilities $\widetilde{f}\left(k\right)$ can be approximated by the normalized histogram $H(k)/N$.
Equation:

$$\widetilde{f}\left(k\right) \approx \frac{H(k)}{N}$$

Using [link], we may then approximate the density function $f_X\left(x\right)$ within $bin(k)$ by
Equation:

$$f_X\left(x\right) \approx \frac{H(k)}{N\Delta} \quad \text{for } x \in bin\left(k\right) \ .$$

Notice this estimate is a staircase function of x which is constant over each interval $bin(k)$. It can also easily be verified that this density estimate integrates to 1.

Exercise

Let U be a uniformly distributed random variable on the interval [0,1] with the following cumulative probability distribution, $F_U(u)$:
Equation:

$$F_U(u) = \begin{cases} 0, & \text{if } u < 0 \\ u, & \text{if } 0 \leq u \leq 1 \\ 1, & \text{if } u > 1 \end{cases}$$

We can calculate the cumulative probability distribution for the new random variable $X = U^{\frac{1}{3}}$.
Equation:

$$\begin{aligned} F_X(x) &= P(X \leq x) \\ &= P\left(U^{\frac{1}{3}} \leq x\right) \\ &= P\left(U \leq x^3\right) \\ &= F_U(u)|_{u=x^3} \\ &= \begin{cases} 0, & \text{if } x < 0 \\ x^3, & \text{if } 0 \leq x \leq 1 \\ 1, & \text{if } x > 1 \end{cases} \end{aligned}$$

Plot $F_X(x)$ for $x \in [0,1]$. Also, analytically calculate the probability density $f_X(x)$, and plot it for $x \in [0,1]$.

Using $L = 20$, $x_0 = 0$ and $x_L = 1$, use Matlab to compute $\widetilde{f}(k)$, the probability of X falling into $bin(k)$.

Note: Use the fact that $\widetilde{f}(k) = F_X(x_k) - F_X(x_{k-1})$.

Plot $\widetilde{f}(k)$ for $k = 1, \cdots, L$ using the `stem` function.

INLAB REPORT

1. Submit your plots of $F_X(x)$, $f_X(x)$ and $\widetilde{f}(k)$. Use `stem` to plot $\widetilde{f}(k)$, and put all three plots on a single figure using `subplot`.
2. Show (mathematically) how $f_X(x)$ and $\widetilde{f}(k)$ are related.

Generate 1000 samples of a random variable U that is uniformly distributed between 0 and 1 (using the `rand` command). Then form the random vector X by computing $X = U^{\frac{1}{3}}$.

Use the Matlab function `hist` to plot a normalized histogram for your samples of X, using 20 bins uniformly spaced on the interval $[0, 1]$.

Note: Use the Matlab command `H=hist(X,(0.5:19.5)/20)` to obtain the histogram, and then normalize `H`.

Use the `stem` command to plot the normalized histogram $H(k)/N$ and $\widetilde{f}(k)$ together on the same figure using `subplot`.

INLAB REPORT

1. Submit your two stem plots of $H(k)/N$ and $\widetilde{f}(k)$. How do these plots compare?
2. Discuss the tradeoffs (advantages and the disadvantages) between selecting a very large or very small bin-width.

Lab 7b - Discrete-Time Random Processes (part 2)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Bivariate Distributions

In this section, we will study the concept of a bivariate distribution. We will see that bivariate distributions characterize how two random variables are related to each other. We will also see that correlation and covariance are two simple measures of the dependencies between random variables, which can be very useful for analyzing both random variables and random processes.

Background on Bivariate Distributions

Sometimes we need to account for not just one random variable, but several. In this section, we will examine the case of two random variables–the so called **bivariate** case–but the theory is easily generalized to accommodate more than two.

The random variables X and Y have cumulative distribution functions (CDFs) $F_X(x)$ and $F_Y(y)$, also known as **marginal** CDFs. Since there may be an interaction between X and Y, the marginal statistics may not fully describe their behavior. Therefore we define a **bivariate**, or **joint** CDF as
Equation:

$$F_{X,Y}(x,y) = P(X \leq x, Y \leq y).$$

If the joint CDF is sufficiently “smooth”, we can define a joint probability density function,
Equation:

$$f_{X,Y}(x,y) = \frac{\partial^2}{\partial x \partial y} F_{X,Y}(x,y).$$

Conversely, the joint probability density function may be used to calculate the joint CDF:
Equation:

$$F_{X,Y}\left(x, y\right) = \int_{-\infty}^{y} \int_{-\infty}^{x} f_{X,Y}\left(s, t\right) ds\, dt.$$

The random variables X and Y are said to be **independent** if and only if their joint CDF (or PDF) is a separable function, which means
Equation:

$$f_{X,Y}\left(x, y\right) = f_X\left(x\right) f_Y\left(y\right)$$

Informally, independence between random variables means that one random variable does not tell you anything about the other. As a consequence of the definition, if X and Y are independent, then the product of their expectations is the expectation of their product.
Equation:

$$E[XY] = E[X]E[Y]$$

While the joint distribution contains all the information about X and Y, it can be very complex and is often difficult to calculate. In many applications, a simple measure of the dependencies of X and Y can be very useful. Three such measures are the **correlation**, **covariance**, and the **correlation coefficient**.

- Correlation
  Equation:

  $$E\left[XY\right] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} xy f_{X,Y}\left(x, y\right) dx\, dy$$

- Covariance
  Equation:

  $$E\left[\left(X - \mu_X\right)\left(Y - \mu_Y\right)\right] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \left(x - \mu_X\right)\left(y - \mu_Y\right) f_{X,Y}\left(x, y\right) dx dy$$

- Correlation coefficient
  Equation:

$$\rho_{XY} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} = \frac{E[XY] - \mu_X \mu_Y}{\sigma_X \sigma_Y}$$

If the correlation coefficient is 0, then X and Y are said to be **uncorrelated**. Notice that independence implies uncorrelatedness, however the converse is not true.

Samples of Two Random Variables

In the following experiment, we will examine the relationship between the scatter plots for pairs of random samples (X_i, Z_i) and their correlation coefficient. We will see that the correlation coefficient determines the shape of the scatter plot.

Let X and Y be independent Gaussian random variables, each with mean 0 and variance 1. We will consider the correlation between X and Z, where Z is equal to the following:

1. **Equation:**

$$Z = Y$$

2. **Equation:**

$$Z = (X + Y)/2$$

3. **Equation:**

$$Z = (4*X + Y)/5$$

4. **Equation:**

$$Z = (99*X + Y)/100$$

Notice that since Z is a linear combination of two Gaussian random variables, Z will also be Gaussian.

Use Matlab to generate 1000 i.i.d. samples of X, denoted as X_1, X_2, ..., X_{1000}. Next, generate 1000 i.i.d. samples of Y, denoted as Y_1, Y_2, ..., Y_{1000}. For each of the four choices of Z, perform the following tasks:

1. Use [link] to analytically calculate the correlation coefficient ρ_{XZ} between X and Z. Show all of your work. Remember that independence between X and Y implies that $E[XY] = E[X]E[Y]$. Also remember that X and Y are zero-mean and unit variance.
2. Create samples of Z using your generated samples of X and Y.
3. Generate a scatter plot of the ordered pair of samples (X_i, Z_i). Do this by plotting points (X_1, Z_1), (X_2, Z_2), ..., (X_{1000}, Z_{1000}). To plot points without connecting them with lines, use the '.' format, as in `plot(X,Z,'.')`. Use the command `subplot(2,2,n)` `(n=1,2,3,4)` to plot the four cases for Z in the same figure. Be sure to label each plot using the `title` command.
4. Empirically compute an estimate of the correlation coefficient using your samples X_i and Z_i and the following formula.
   Equation:

$$\widehat{\rho}_{XZ} = \frac{\sum_{i=1}^{N} \left(X_i - \widehat{\mu}_X\right)\left(Z_i - \widehat{\mu}_Z\right)}{\sqrt{\sum_{i=1}^{N} \left(X_i - \widehat{\mu}_X\right)^2 \sum_{i=1}^{N} \left(Z_i - \widehat{\mu}_Z\right)^2}}$$

INLAB REPORT

1. Hand in your derivations of the correlation coefficient ρ_{XZ} along with your numerical estimates of the correlation coefficient $\widehat{\rho}_{XZ}$.
2. Why are ρ_{XZ} and $\widehat{\rho}_{XZ}$ not exactly equal?
3. Hand in your scatter plots of (X_i, Z_i) for the four cases. Note the theoretical correlation coefficient ρ_{XZ} on each plot.
4. Explain how the scatter plots are related to ρ_{XZ}.

Autocorrelation for Filtered Random Processes

In this section, we will generate **discrete-time random processes** and then analyze their behavior using the correlation measure introduced in the previous section.

Background

A **discrete-time random process** X_n is simply a sequence of random variables. So for each n, X_n is a random variable.

The **autocorrelation** is an important function for characterizing the behavior of random processes. If X is a **wide-sense stationary** (WSS) random process, the autocorrelation is defined by
Equation:

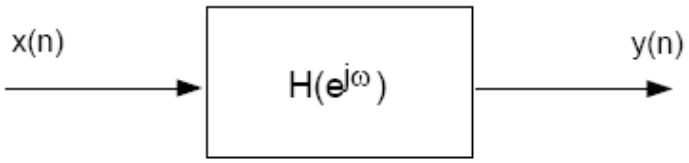
$$r_{XX}(m) = E[X_n X_{n+m}] \quad m = \cdots, -1, 0, 1, \cdots \ .$$

Note that for a WSS random process, the autocorrelation does not vary with n. Also, since $E[X_n X_{n+m}] = E[X_{n+m} X_n]$, the autocorrelation is an even function of the "lag" value m.

Intuitively, the autocorrelation determines how strong a relation there is between samples separated by a lag value of m. For example, if X is a sequence of independent identically distributed (i.i.d.) random variables each with zero mean and variance σ_X^2, then the autocorrelation is given by
Equation:

$$\begin{aligned} r_{XX}(m) &= E[X_n X_{n+m}] \\ &= \begin{cases} E[X_n] E[X_{n+m}] & \text{if } m \neq 0 \\ E\left[X_n^2\right] & \text{if } m = 0 \end{cases} \\ &= \sigma_X^2 \delta(m) \ . \end{aligned}$$

We use the term **white** or **white noise** to describe this type of random process. More precisely, a random process is called **white** if its values X_n and X_{n+m} are uncorrelated for every $m \neq 0$.



A LTI system diagram

If we run a white random process X_n through an LTI filter as in [link], the output random variables Y_n may become correlated. In fact, it can be shown that the output autocorrelation $r_{YY}(m)$ is related to the input autocorrelation $r_{XX}(m)$ through the filter's impulse response $h(m)$.
Equation:

$$r_{YY}(m) = h(m)*h(-m)*r_{XX}(m)$$

Experiment

Consider a white Gaussian random process X_n with mean 0 and variance 1 as input to the following filter.
Equation:

$$y(n) = x(n) - x(n-1) + x(n-2)$$

Calculate the theoretical autocorrelation of Y_n using [link] and [link]. Show all of your work.

Generate 1000 independent samples of a Gaussian random variable X with mean 0 and variance 1. Filter the samples using [link]. We will denote the filtered signal Y_i, $i = 1, 2, \cdots, 1000$.

Draw 4 scatter plots using the form `subplot(2,2,n)`, $(n = 1, 2, 3, 4)$. The first scatter plot should consist of points, (Y_i, Y_{i+1}), $(i = 1, 2, \cdots, 900)$. Notice that this correlates samples that are separated by a lag of “1”. The other 3 scatter

plots should consist of the points (Y_i, Y_{i+2}), (Y_i, Y_{i+3}), (Y_i, Y_{i+4}), $(i = 1, 2, \cdots, 900)$, respectively. What can you deduce about the random process from these scatter plots?

For real applications, the theoretical autocorrelation may be unknown. Therefore, $r_{YY}(m)$ may be estimated by the **sample autocorrelation**, $r'_{YY}(m)$ defined by
Equation:

$$r'_{YY}(m) = \frac{1}{N - |m|} \sum_{n=0}^{N-|m|-1} Y(n)Y(n + |m|) \quad -(N-1) \leq m \leq N-1$$

where N is the number of samples of Y.

Use Matlab to calculate the sample autocorrelation of Y_n using [link]. Plot both the theoretical autocorrelation $r_{YY}(m)$, and the sample autocorrelation $r'_{YY}(m)$ versus m for $-20 \leq m \leq 20$. Use `subplot` to place them in the same figure. Does [link] produce a reasonable approximation of the true autocorrelation?

INLAB REPORT
For the filter in [link],

1. Show your derivation of the theoretical output autocorrelation, $r_{YY}(m)$.
2. Hand in the four scatter plots. Label each plot with the corresponding theoretical correlation, from $r_{YY}(m)$. What can you conclude about the output random process from these plots?
3. Hand in your plots of $r_{YY}(m)$ and $r'_{YY}(m)$ versus m. Does [link] produce a reasonable approximation of the true autocorrelation? For what value of m does $r_{YY}(m)$ reach its maximum? For what value of m does $r'_{YY}(m)$ reach its maximum?
4. Hand in your Matlab code.

Correlation of Two Random Processes

Background

The **cross-correlation** is a function used to describe the correlation between two separate random processes. If X and Y are jointly WSS random processes, the cross-correlation is defined by
Equation:

$$c_{XY}(m) = E[X_n Y_{n+m}] \quad m = \cdots, -1, 0, 1, \cdots \ .$$

Similar to the definition of the sample autocorrelation introduced in the previous section, we can define the **sample cross-correlation** for a pair of data sets. The **sample cross-correlation** between two finite random sequences X_n and Y_n is defined as
Equation:

$$c'_{XY}(m) = \frac{1}{N-m} \sum_{n=0}^{N-m-1} X(n) Y(n+m) \quad 0 \leq m \leq N-1$$

Equation:

$$c'_{XY}(m) = \frac{1}{N-|m|} \sum_{n=|m|}^{N-1} X(n) Y(n+m) \quad 1-N \leq m < 0$$

where N is the number of samples in **each** sequence. Notice that the cross-correlation is not an even function of m. Hence a two-sided definition is required.

Cross-correlation of signals is often used in applications of sonar and radar, for example to estimate the distance to a target. In a basic radar set-up, a zero-mean signal $X(n)$ is transmitted, which then reflects off a target after traveling for $D/2$ seconds. The reflected signal is received, amplified, and then digitized to form $Y(n)$. If we summarize the attenuation and amplification of the received signal by the constant α, then
Equation:

$$Y(n) = \alpha X(n-D) + W(n)$$

where $W(n)$ is additive noise from the environment and receiver electronics.

In order to compute the distance to the target, we must estimate the delay D. We can do this using the cross-correlation. The cross-correlation c_{XY} can be calculated by substituting [link] into [link].
Equation:

$$\begin{aligned} c_{XY}(m) &= E[X(n)Y(n+m)] \\ &= E[X(n)(\alpha X(n-D+m)+W(n+m))] \\ &= \alpha E[X(n)X(n-D+m)]+E[X(n)]E[W(n+m)] \\ &= \alpha E[X(n)X(n-D+m)] \end{aligned}$$

Here we have used the assumptions that $X(n)$ and $W(n+m)$ are uncorrelated and zero-mean. By applying the definition of autocorrelation, we see that
Equation:

$$c_{XY}(m) = \alpha r_{XX}(m-D)$$

Because $r_{XX}(m-D)$ reaches its maximum when $m=D$, we can find the delay D by searching for a peak in the cross correlation $c_{XY}(m)$. Usually the transmitted signal $X(n)$ is designed so that $r_{XX}(m)$ has a large peak at $m=0$.

Experiment

Download the file radar.mat for the following section.

Using [link] and [link], write a Matlab function `C=CorR(X,Y,m)` to compute the sample cross-correlation between two discrete-time random processes, X and Y, for a single lag value m.

To test your function, generate two length 1000 sequences of zero-mean Gaussian random variables, denoted as X_n and Z_n. Then compute the new sequence $Y_n = X_n + Z_n$. Use `CorR` to calculate the sample cross-correlation between X and Y for lags $-10 \leq m \leq 10$. Plot your cross-correlation function.

INLAB REPORT

1. Submit your plot for the cross-correlation between X and Y. Label the m-axis with the corresponding lag values.
2. Which value of m produces the largest cross-correlation? Why?
3. Is the cross-correlation function an even function of m? Why or why not?
4. Hand in the code for your CorR function.

Next we will do an experiment to illustrate how cross-correlation can be used to measure time delay in radar applications. Down load the MAT file radar.mat and load it using the command `load radar`. The vectors **trans** and **received** contain two signals corresponding to the transmitted and received signals for a radar system. First compute the autocorrelation of the signal **trans** for the lags $-100 \leq m \leq 100$. (Hint: Use your `CorR` function.)

Next, compute the sample cross-correlation between the signal **trans** and **received** for the range of lag values $-100 \leq m \leq 100$, using your $CorR$ function. Determine the delay D.

INLAB REPORT

1. Plot the transmitted signal and the received signal on a single figure using `subplot`. Can you estimate the delay D by a visual inspection of the received signal?
2. Plot the sample autocorrelation of the transmitted signal, $r'_{XX}(m)$ vs. m for $-100 \leq m \leq 100$.
3. Plot the sample cross-correlation of the transmitted signal and the received signal, $c'_{XY}(m)$ vs. m for $-100 \leq m \leq 100$.
4. Determine the delay D from the sample correlation. How did you determine this?

Lab 7c - Power Spectrum Estimation

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

In the first and second weeks of this experiment, we introduced methods of statistically characterizing random processes. The sample autocorrelation and cross correlation are examples of “time domain” characterizations of random signals. In many applications, it can also be useful to get the frequency domain characteristics of a random process. Examples include detection of sinusoidal signals in noise, speech recognition and coding, and range estimation in radar systems.

In this week, we will introduce methods to estimate the **power spectrum** of a random signal given a finite number of observations. We will examine the effectiveness of the **periodogram** for spectrum estimation, and introduce the **spectrogram** to characterize a nonstationary random processes.

Power Spectrum Estimation

In this section, you will estimate the power spectrum of a stationary discrete random process. The power spectrum is defined as
Equation:

$$S_{xx}(\omega) = \lim_{N\to\infty} E\left[\frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-j\omega n}\right]^2$$

There are 4 steps for calculating a power spectrum:

1. Select a window of length N and generate a finite sequence $x(0), x(1), \cdots, x(N-1)$.

2. Calculate the DTFT of the windowed sequence $x(n)$, $(n = 0, 1, \cdots, N-1)$, square the magnitude of the DTFT and divide it by the length of the sequence.
3. Take the expectation with respect to x.
4. Let the length of the window go to infinity.

In real applications, we can only approximate the power spectrum. Two methods are introduced in this section. They are the **periodogram** and the **averaged periodogram**.

Periodogram

Click here for help on the fft command and the random command.

The periodogram is a simple and common method for estimating a power spectrum. Given a finite duration discrete random sequence $x(n)$, $(n = 0, 1, \cdots, N-1)$, the periodogram is defined as
Equation:

$$P_{xx}\left(\omega\right) = \frac{1}{N}\left|X(\omega)\right|^2 = \frac{1}{N}\left|\sum_{n=0}^{N-1} x\left(n\right)e^{-j\omega n}\right|^2$$

where $X(\omega)$ is the Discrete Time Fourier Transform (DTFT) of $x(n)$.

The periodogram $P_{xx}\left(\omega\right)$ can be computed using the Discrete Fourier Transformation (DFT), which in turn can be efficiently computed by the Fast Fourier Transformation (FFT). If $x(n)$ is of length N, you can compute an N-point DFT.
Equation:

$$P_{xx}\left(\omega_k\right) = \frac{1}{N}\left|\sum_{n=0}^{N-1} x\left(n\right)e^{-j\omega_k n}\right|^2$$

Equation:

$$\omega_k = \frac{2\pi k}{N}, \quad k = 0, 1, \cdots, N-1$$

Using [link] and [link], write a Matlab function called `Pgram` to calculate the periodogram. The syntax for this function should be

```
[P,w] = Pgram(x)
```

where x is a discrete random sequence of length N. The outputs of this command are P, the samples of the periodogram, and w, the corresponding frequencies of the samples. Both P and w should be vectors of length N.

Now, let x be a Gaussian (Normal) random variable with mean 0 and variance 1. Use Matlab function `random` or `randn` to generate 1024 i.i.d. samples of x, and denote them as x_0, x_1, ..., x_{1023}. Then filter the samples of x with the filter which obeys the following difference equation
Equation:

$$y(n) = 0.9y(n-1) + 0.3x(n) + 0.24x(n-1) \ .$$

Denote the output of the filter as $y_0, y_1, \cdots, y_{1023}$.

Note: The samples of $x(n)$ and $y(n)$ will be used in the following sections. Please save them.

Use your `Pgram` function to estimate the power spectrum of $y(n)$, $P_{yy}(\omega_k)$. Plot $P_{yy}(\omega_k)$ vs. ω_k.

Next, estimate the power spectrum of y, $P_{yy}(\omega_k)$ using $1/4$ of the samples of y. Do this only using samples y_0, y_1, $\cdots$, y_{255}. Plot $P_{yy}(\omega_k)$ vs. ω_k.

INLAB REPORT

1. Hand in your labeled plots and your `Pgram` code.
2. Compare the two plots. The first plot uses 4 times as many samples as the second one. Is the first one a better estimation than the second one? Does the first give you a smoother estimation?
3. Judging from the results, when the number of samples of a discrete random variable becomes larger, will the estimated power spectrum be smoother?

Averaged Periodogram

The periodogram is a simple method, but it does not yield very good results. To get a better estimate of the power spectrum, we will introduce Bartlett's method, also known as **the averaged periodogram**. This method has three steps. Suppose we have a length-N sequence $x(n)$.

1. Subdivide $x(n)$ into K nonoverlapping segments of length M. Denote the i^{th} segment as $x_i\left(n\right)$.
   Equation:

$$x_i\left(n\right) = x\left(n + iM\right), \quad i = 0, 1, \cdots, K-1, \quad n = 0, 1, \cdots, M-1$$

2. For each segment $x_i\left(n\right)$, compute its periodogram
   Equation:

$$P_{x_i x_i}^{(i)}\left(\omega_k\right) = \frac{1}{M} \sum_{n=0}^{M-1} x_i\left(n\right) e^{-j\omega_k n}{}^2$$

   Equation:

$$\omega_k = \frac{2\pi k}{M}$$

   where $k = 0, 1, \cdots, M-1$ and $i = 0, 1, \cdots, K-1$.
3. Average the periodograms over all K segments to obtain the averaged periodogram, $P_{xx}^{A}\left(k\right)$,
   Equation:

$$P_{xx}^{A}(\omega_k) = \frac{1}{K}\sum_{i=0}^{K-1} P_{x_i x_i}^{(i)}(\omega_k)$$

Equation:

$$\omega_k = \frac{2\pi k}{M}$$

where $k = 0, 1, \cdots, M-1$.

Write a Matlab function called `AvPgram` to calculate the averaged periodogram, using the above steps. The syntax for this function should be

```
[P,w] = AvPgram(x, K)
```

where x is a discrete random sequence of length N and the K is the number of nonoverlapping segments. The outputs of this command are P, the samples of the averaged periodogram, and w, the corresponding frequencies of the samples. Both P and w should be vectors of length M where N=KM. You may use your `Pgram` function.

Note: The command

```
A=reshape(x,M,K)
```

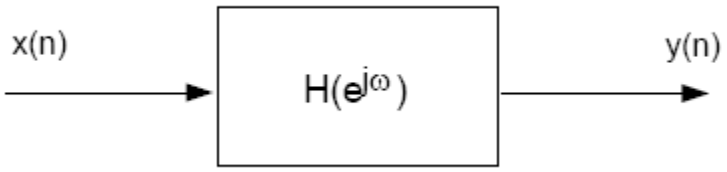
will orient length M segments of the vector x into K columns of the matrix A.

Use your Matlab function `AvPgram` to estimate the power spectrum of $y(n)$ which was generated in the previous section. Use all 1024 samples of $y(n)$, and let $K = 16$. Plot P vs. w.

INLAB REPORT

1. Submit your plot and your `AvPgram` code.
2. Compare the power spectrum that you estimated using the averaged periodogram with the one you calculated in the previous section using the standard periodogram. What differences do you observe? Which do you prefer?

Power Spectrum and LTI Systems



An LTI system

Consider a linear time-invariant system with frequency response $H\left(e^{j\omega}\right)$, where $S_{xx}\left(\omega\right)$ is the power spectrum of the input signal, and $S_{yy}\left(\omega\right)$ is the power spectrum of the output signal. It can be shown that these quantities are related by

Equation:

$$S_{yy}\left(\omega\right) = \left| H\left(e^{j\omega}\right) \right|^2 S_{xx}\left(\omega\right) \ .$$

In the "Periodogram" section, the sequence $y(n)$ was generated by filtering an i.i.d. Gaussian (mean=0, variance=1) sequence $x(n)$, using the filter in [link]. By hand, calculate the power spectrum $S_{xx}\left(\omega\right)$ of $x(n)$, the frequency response of the filter, $H\left(e^{j\omega}\right)$, and the power spectrum $S_{yy}\left(\omega\right)$ of $y(n)$.

Note: In computing $S_{xx}(\omega)$, use the fact that $|ab|^2 = ab^*$.

Plot $S_{yy}(\omega)$ vs. ω, and compare it with the plots from the "Periodogram" and "Averaged Periodogram" sections. What do you observe?

INLAB REPORT

1. Hand in your plot.
2. Submit your analytical calculations for $S_{xx}(\omega)$, $H\left(e^{j\omega}\right)$, $S_{yy}(\omega)$.
3. Compare the theoretical power spectrum of $S_{yy}(\omega)$ with the power spectrum estimated using the standard periodogram. What can you conclude?
4. Compare the theoretical power spectrum of $S_{yy}(\omega)$ with the power spectrum estimated using the averaged periodogram. What can you conclude?

Estimate the Power Spectrum of a Speech Signal

Download the file speech.au for this section. For help on the following Matlab topics select the corresponding link: how to load and play audio signals and specgram function.

The methods used in the last two sections can only be applied to stationary random processes. However, most signals in nature are not stationary. For a nonstationary random process, one way to analyze it is to subdivide the signal into segments (which may be overlapping) and treat each segment as a stationary process. Then we can calculate the power spectrum of each segment. This yields what we call a **spectrogram**.

While it is debatable whether or not a speech signal is actually random, in many applications it is necessary to model it as being so. In this section, you are going to use the Matlab command `specgram` to calculate the spectrogram of a speech signal. Read the help for the specgram function. Find out what the command does, and how to calculate and draw a spectrogram.

Draw the spectrogram of the speech signal in speech.au. When using the `specgram` command with no output arguments, the absolute value of the spectrogram will be plotted. Therefore you can use

```
speech=auread('speech.au');
```

to read the speech signal and use

```
specgram(speech);
```

to draw the spectrogram.

INLAB REPORT

1. Hand in your spectrogram plot.
2. Describe the information that the spectrogram is giving you.

Lab 8 - Number Representation and Quantization

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This lab presents two important concepts for working with digital signals. The first section discusses how numbers are stored in memory. Numbers may be either in fixed point or floating point format. Integers are often represented with fixed point format. Decimals, and numbers that may take on a very large range of values would use floating point. The second issue of numeric storage is quantization. All analog signals that are processed on the computer must first be quantized. We will examine the errors that arise from this operation, and determine how different levels of quantization affect a signal's quality. We will also look at two types of quantizers. The **uniform quantizer** is the simpler of the two. The **Max quantizer**, is optimal in that it minimizes the mean square error between the original and quantized signals.

Review of number representations

There are two types of numbers that a computer can represent: integers and decimals. These two numbers are stored quite differently in memory. Integers (e.g. 27, 0, -986) are usually stored in fixed point format, while decimals (e.g. 12.34, -0.98) most often use floating point format. Most integer representations use four bytes of memory; floating point values usually require eight.

There are different conventions for encoding fixed point binary numbers because of the different ways of representing negative numbers. Three types of fixed point formats that accommodate negative integers are **sign-magnitude**, **one's-complement**, and **two's-complement**. In all three of these "signed" formats, the first bit denotes the sign of the number: 0 for positive, and 1 for negative. For positive numbers, the magnitude simply

follows the first bit. Negative numbers are handled differently for each format.

Of course, there is also an **unsigned** data type which can be used when the numbers are known to be non-negative. This allows a greater range of possible numbers since a bit isn't wasted on the negative sign.

Sign-magnitude representation

Sign-magnitude notation is the simplest way to represent negative numbers. The magnitude of the negative number follows the first bit. If an integer was stored as one byte, the range of possible numbers would be -127 to 127.

The value +27 would be represented as

0 0 0 1 1 0 1 1 .

The number -27 would represented as

1 0 0 1 1 0 1 1 .

One's-complement

To represent a negative number, the complement of the bits for the positive number with the same magnitude are computed. The positive number 27 in one's-complement form would be written as

0 0 0 1 1 0 1 1 ,

but the value -27 would be represented as

1 1 1 0 0 1 0 0 .

Two's-complement

The problem with each of the above notations is that two different values represent zero. Two's-complement notation is a revision to one's-complement that solves this problem. To form negative numbers, the positive number is subtracted from a certain binary number. This number has a one in the most significant bit (MSB), followed by as many zeros as there are bits in the representation. If 27 was represented by an eight-bit integer, -27 would be represented as:

```
   1 0  0 0 0 0 0 0 0
   - 0  0 0 1 1 0 1 1
 =   1  1 1 0 0 1 0 1
```

Notice that this result is one **plus** the one's-complement representation for -27 (modulo-2 addition). What about the second value of 0? That representation is

1 0 0 0 0 0 0 0 .

This value equals -128 in two's-complement notation!

```
   1 0  0 0 0 0 0 0 0
   - 1  0 0 0 0 0 0 0
 =   1  0 0 0 0 0 0 0
```

The value represented here is -128; we know it is negative, because the result has a 1 in the MSB. Two's-complement is used because it can represent one extra negative value. More importantly, if the sum of a series of two's-complement numbers is within the range, overflows that occur during the summation will not affect the final answer! The range of an 8-bit two's complement integer is [-128,127].

Floating Point

Floating point notation is used to represent a much wider range of numbers. The tradeoff is that the resolution is variable: it **decreases** as the magnitude

of the number increases. In the fixed point examples above, the resolution was fixed at 1. It is possible to represent decimals with fixed point notation, but for a fixed word length any increase in resolution is matched by a decrease in the range of possible values.

A floating point number, F, has two parts: a **mantissa**, M, and an **exponent**, E.
Equation:

$$F = M*2^E$$

The mantissa is a signed fraction, which has a power of two in the denominator. The exponent is a signed integer, which represents the power of two that the mantissa must be multiplied by. These signed numbers may be represented with any of the three fixed-point number formats. The IEEE has a standard for floating point numbers (IEEE 754). For a 32-bit number, the first bit is the mantissa's sign. The exponent takes up the next 8 bits (1 for the sign, 7 for the quantity), and the mantissa is contained in the remaining 23 bits. The range of values for this number is ($-1.18*10^{-38}, 3.40*10^{38}$).

To add two floating point numbers, the exponents must be the same. If the exponents are different, the mantissa is adjusted until the exponents match. If a very small number is added to a large one, the result may be the same as the large number! For instance, if $0.15600\cdots0*2^{30}$ is added to $0.62500\cdots0*2^{-3}$, the second number would be converted to $0.0000\cdots0*2^{30}$ before addition. Since the mantissa only holds 23 binary digits, the decimal digits 625 would be lost in the conversion. In short, the second number is rounded down to zero. For multiplication, the two exponents are added and the mantissas multiplied.

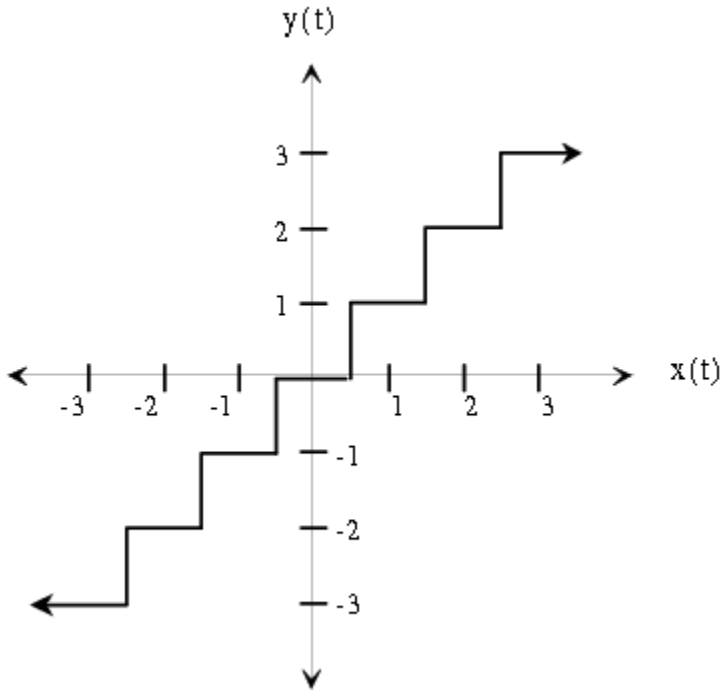
Quantization

Introduction

Quantization is the act of rounding off the value of a signal or quantity to certain discrete levels. For example, digital scales may round off weight to the nearest gram. Analog voltage signals in a control system may be rounded off to the nearest volt before they enter a digital controller. Generally, all numbers need to be quantized before they can be represented in a computer.

Digital images are also quantized. The gray levels in a black and white photograph must be quantized in order to store an image in a computer. The “brightness" of the photo at each pixel is assigned an integer value between 0 and 255 (typically), where 0 corresponds to black, and 255 to white. Since an 8-bit number can represent 256 different values, such an image is called an “8-bit grayscale" image. An image which is quantized to just 1 bit per pixel (in other words only black and white pixels) is called a **halftone** image. Many printers work by placing, or not placing, a spot of colorant on the paper at each point. To accommodate this, an image must be halftoned before it is printed.

Quantization can be thought of as a functional mapping $y = f(x)$ of a real-valued input to a discrete-valued output. An example of a quantization function is shown in [link], where the x-axis is the input value, and the y-axis is the quantized output value.



Input-output relation for a 7-level uniform quantizer.

Quantization and Compression

Quantization is sometimes used for compression. As an example, suppose we have a digital image which is represented by 8 different gray levels: [0 31 63 95 159 191 223 255]. To directly store each of the image values, we need at least 8-bits for each pixel since the values range from 0 to 255. However, since the image only takes on 8 different values, we can assign a different 3-bit integer (a code) to represent each pixel: [000 001 ... 111]. Then, instead of storing the actual gray levels, we can store the 3-bit code for each pixel. A look-up table, possibly stored at the beginning of the file, would be used to decode the image. This lowers the cost of an image considerably: less hard drive space is needed, and less bandwidth is required to transmit the image (i.e. it downloads quicker). In practice, there

are much more sophisticated methods of compressing images which rely on quantization.

Image Quantization

Download the file fountainbw.tif for the following section.

The image in `fountainbw.tif` is an 8-bit grayscale image. We will investigate what happens when we quantize it to fewer bits per pixel (b/pel). Load it into Matlab and display it using the following sequence of commands:

```
y = imread('fountainbw.tif');

image(y);

colormap(gray(256));

axis('image');
```

The image array will initially be of type `uint8`, so you will need to convert the image matrix to type `double` before performing any computation. Use the command `z=double(y)` for this.

There is an easy way to uniformly quantize a signal. Let
Equation:

$$\Delta = \frac{Max(X) - Min(X)}{N - 1}$$

where X is the signal to be quantized, and N is the number of quantization levels. To force the data to have a uniform quantization step of Δ,

- Subtract Min(X) from the data and divide the result by Δ.
- Round the data to the nearest integer.

- Multiply the rounded data by Δ and add Min(X) to convert the data back to its original scale.

Write a Matlab function `Y = Uquant(X,N)` which will uniformly quantize an input array X (either a vector or a matrix) to N discrete levels. Use this function to quantize the fountain image to 7 b/pel, 6, 5, 4, 3, 2, 1 b/pel, and observe the output images.

Note: Remember that with b bits, we can represent $N = 2^b$ gray levels.

Print hard copies of only the 7, 4, 2, and 1 b/pel images, as well as the original.

INLAB REPORT

1. Describe the artifacts (errors) that appear in the image as the number of bits is lowered?
2. Note the number of b/pel at which the image quality noticeably deteriorates.
3. Hand in the printouts of the above four quantized images and the original.
4. Compare each of these four quantized images to the original.

Audio Quantization

Download the files speech.au and [missing_resource: music.au]

If an audio signal is to be coded, either for compression or for digital transmission, it must undergo some form of quantization. Most often, a general technique known as **vector quantization** is employed for this task, but this technique must be tailored to the specific application so it will not be addressed here. In this exercise, we will observe the effect of uniformly quantizing the samples of two audio signals.

Down load the audio files speech.au and [missing_resource: music.au]`Uquant` function to quantize each of these signals to 7, 4, 2 and 1 bits/sample. Listen to the original and quantized signals and answer the following questions:

- For each signal, describe the change in quality as the number of b/sample is reduced?
- For each signal, is there a point at which the signal quality deteriorates drastically? At what point (if any) does it become incomprehensible?
- Which signal's quality deteriorates faster as the number of levels decreases?
- Do you think 4 b/sample is acceptable for telephone systems? ... 2 b/sample?

Use `subplot` to plot in the same figure, the four quantized speech signals over the index range 7201:7400. Generate a similar figure for the music signal, using the same indices. Make sure to use `orient tall` before printing these out.

Note: Hand in answers to the above questions, and the two Matlab figures.

Error Analysis

As we have clearly observed, quantization produces errors in a signal. The most effective methods for analysis of the error turn out to be probabilistic. In order to apply these methods, however, one needs to have a clear understanding of the error signal's statistical properties. For example, can we assume that the error signal is white noise? Can we assume that it is uncorrelated with the quantized signal? As you will see in this exercise, both of these are good assumptions if the quantization intervals are small compared with sample-to-sample variations in the signal.

If the original signal is X, and the quantized signal is Y, the error signal is defined by the following:

Equation:

$$E = Y - X$$

Compute the error signal for the quantized speech for 7, 4, 2 and 1 b/sample.

When the spacing, Δ, between quantization levels is sufficiently small, a common statistical model for the error is a uniform distribution from $\frac{-\Delta}{2}$ to $\frac{\Delta}{2}$. Use the command `hist(E,20)` to generate 20-bin histograms for each of the four error signals. Use `subplot` to place the four histograms in the same figure.

INLAB REPORT

1. Hand in the histogram figure.
2. How does the number of quantization levels seem to affect the shape of the distribution?
3. Explain why the error histograms you obtain might not be uniform?

Next we will examine correlation properties of the error signal. First compute and plot an estimate of the autocorrelation function for each of the four error signals using the following commands:

```
[r,lags] = xcorr(E,200,'unbiased');
```

```
plot(lags,r)
```

Now compute and plot an estimate of the cross-correlation function between the quantized speech `Y` and each error signal `E` using

```
[c,lags] = xcorr(E,Y,200,'unbiased');
```

```
plot(lags,c)
```

INLAB REPORT

1. Hand in the autocorrelation and cross-correlation estimates.
2. Is the autocorrelation influenced by the number of quantization levels? Do samples in the error signal appear to be correlated with each other?
3. Does the number of quantization levels influence the cross-correlation?

Signal to Noise Ratio

One way to measure the quality of a quantized signal is by the Power Signal-to-Noise Ratio (PSNR). This is defined by the ratio of the power in the quantized speech to power in the noise.
Equation:

$$PSNR = \frac{P_Y}{P_E}$$

In this expression, the noise is the error signal `E`. Generally, this means that a higher PSNR implies a less noisy signal.

From previous labs we know the power of a sampled signal, $x(n)$, is defined by
Equation:

$$P_x = \frac{1}{L}\sum_{n=1}^{L} x^2\,(n)$$

where L is the length of $x(n)$. Compute the PSNR for the four quantized speech signals from the previous section.

In evaluating quantization (or compression) algorithms, a graph called a “rate-distortion curve" is often used. This curve plots **signal distortion** vs. **bit rate**. Here, we can measure the distortion by $\frac{1}{PSNR}$, and determine the bit rate from the number of quantization levels and sampling rate. For

example, if the sampling rate is 8000 samples/sec, and we are using 7 bits/sample, the bit rate is 56 kilobits/sec (kbps).

Assuming that the speech is sampled at 8kHz, plot the rate distortion curve using $\frac{1}{PSNR}$ as the measure of distortion. Generate this curve by computing the PSNR for 7, 6, 5,..., 1 bits/sample. Make sure the axes of the graph are in terms of **distortion** and **bit rate**.

Note: Hand in a list of the 4 PSNR values, and the rate-distortion curve.

Max Quantizer

In this section, we will investigate a different type of quantizer which produces less noise for a fixed number of quantization levels. As an example, let's assume the input range for our signal is [-1,1], but most of the input signal takes on values between [-0.2, 0.2]. If we place more of the quantization levels closer to zero, we can lower the average error due to quantization.

A common measure of quantization error is mean squared error (noise power). The **Max quantizer** is designed to minimize the mean squared error for a given set of training data. We will study how the Max quantizer works, and compare its performance to that of the uniform quantizer which was used in the previous sections.

Derivation

The Max quantizer determines quantization levels based on a data set's probability density function, $f(x)$, and the number of desired levels, N. It minimizes the mean squared error between the original and quantized signals:

Equation:

$$\varepsilon = \sum_{k=1}^{N} \int_{x_k}^{x_{k+1}} (q_k - x)^2 f(x) dx$$

where q_k is the k^{th} quantization level, and x_k is the lower boundary for q_k. The error ε depends on both q_k and x_k. (Note that for the Gaussian distribution, $x_1 = -\infty$, and $x_{N+1} = \infty$.) To minimize ε with respect to q_k, we must take $\frac{\partial \varepsilon}{\partial q_k} = 0$ and solve for q_k:

Equation:

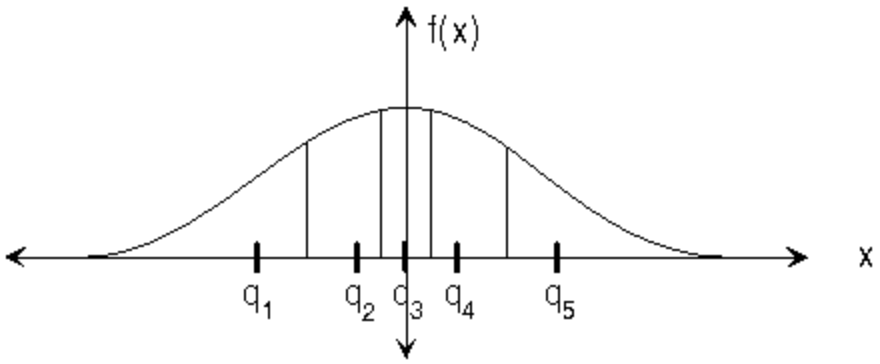
$$q_k = \frac{\int_{x_k}^{x_{k+1}} x f(x) dx}{\int_{x_k}^{x_{k+1}} f(x) dx}$$

We still need the quantization boundaries, x_k. Solving $\frac{\partial \varepsilon}{\partial x_k} = 0$ yields:

Equation:

$$x_k = \frac{q_{k-1} + q_k}{2}$$

This means that each non-infinite boundary is exactly halfway in between the two adjacent quantization levels, and that each quantization level is at the centroid of its region. [link] shows a five-level quantizer for a Gaussian distributed signal. Note that the levels are closer together in areas of higher probability.



Five level Max quantizer for Gaussian-distributed signal.

Implementation, Error Analysis and Comparison

Download the file speech.au for the following section.

In this section we will use Matlab to compute an optimal quantizer, and compare its performance to the uniform quantizer. Since we almost never know the actual probability density function of the data that the quantizer will be applied to, we cannot use equation [link] to compute the optimal quantization levels. Therefore, a numerical optimization procedure is used on a **training set** of data to compute the quantization levels and boundaries which yield the smallest possible error for that set.

Matlab has a built-in function called `lloyds` which performs this optimization. It's syntax is...

```
[partition, codebook] = lloyds(training_set, initial_codebook) ;
```

This function requires two inputs. The first is the training data set, from which it will estimate the probability density function. The second is a vector containing an initial guess of the optimal quantization levels. It

returns the computed optimal boundaries (the “partition”) and quantization levels (the “codebook”).

Since this algorithm minimizes the error with respect to the quantization levels, it is necessary to provide a decent initial guess of the codebook to help ensure a valid result. If the initial codebook is significantly “far" away from the optimal solution, it's possible that the optimization will get trapped in a local minimum, and the resultant codebook may perform quite poorly. In order to make a good guess, we may first estimate the shape of the probability density function of the training set using a histogram. The idea is to divide the histogram into equal “areas" and choose quantization levels as the centers of each of these segments.

First plot a 40-bin histogram of this speech signal using `hist(speech,40)`, and make an initial guess of the four optimal quantization levels. Print out the histogram. Then use the `lloyds` function to compute an optimal 4-level codebook using speech.au as the training set.

Once the optimal codebook is obtained, use the **codebook** and **partition** vectors to quantize the speech signal. This may be done with a **for** loop and **if** statements. Then compute the error signal and PSNR. On the histogram plot, mark where the optimal quantization levels fall along the x-axis.

INLAB REPORT

1. Turn in the histogram plot with the codebook superimposed.
2. Compare the PSNR and sound quality of the uniform- and Max-quantized signals.
3. If the speech signal was uniformly distributed, would the two quantizers be the same? Explain your answer.

Lab 9a - Speech Processing (part 1)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

Speech is an acoustic waveform that conveys information from a speaker to a listener. Given the importance of this form of communication, it is no surprise that many applications of signal processing have been developed to manipulate speech signals. Almost all speech processing applications currently fall into three broad categories: speech recognition, speech synthesis, and speech coding.

Speech recognition may be concerned with the identification of certain words, or with the identification of the speaker. **Isolated word recognition** algorithms attempt to identify individual words, such as in automated telephone services. Automatic speech recognition systems attempt to recognize continuous spoken language, possibly to convert into text within a word processor. These systems often incorporate grammatical cues to increase their accuracy. Speaker identification is mostly used in security applications, as a person's voice is much like a “fingerprint”.

The objective in speech synthesis is to convert a string of text, or a sequence of words, into natural-sounding speech. One example is the Speech Plus synthesizer used by Stephen Hawking (although it unfortunately gives him an American accent). There are also similar systems which read text for the blind. Speech synthesis has also been used to aid scientists in learning about the mechanisms of human speech production, and thereby in the treatment of speech-related disorders.

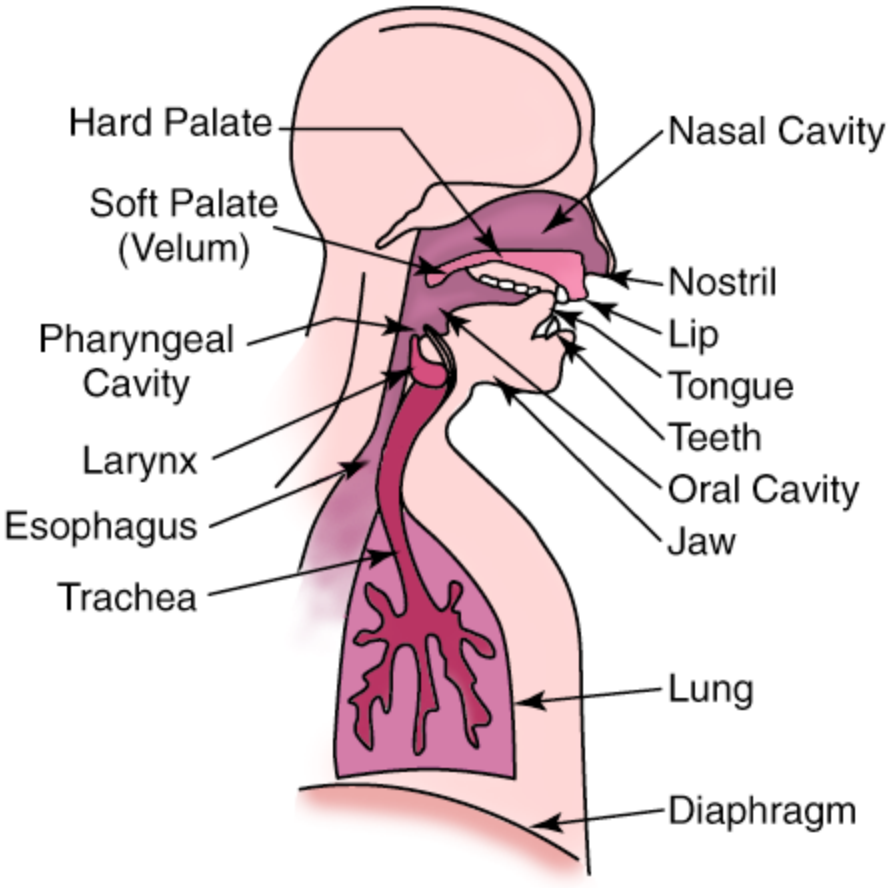
Speech coding is mainly concerned with exploiting certain redundancies of the speech signal, allowing it to be represented in a compressed form. Much of the research in speech compression has been motivated by the need to

conserve bandwidth in communication systems. For example, speech coding is used to reduce the bit rate in digital cellular systems.

In this lab, we will describe some elementary properties of speech signals, introduce a tool known as the **short-time discrete-time Fourier Transform**, and show how it can be used to form a **spectrogram**. We will then use the spectrogram to estimate properties of speech waveforms.

This is the first part of a two-week experiment. During the second week , we will study speech models and linear predictive coding.

Time Domain Analysis of Speech Signals



The Human Speech Production System

Speech Production

Speech consists of acoustic pressure waves created by the voluntary movements of anatomical structures in the human speech production system, shown in [link]. As the diaphragm forces air through the system, these structures are able to generate and shape a wide variety of waveforms. These waveforms can be broadly categorized into **voiced** and **unvoiced speech**.

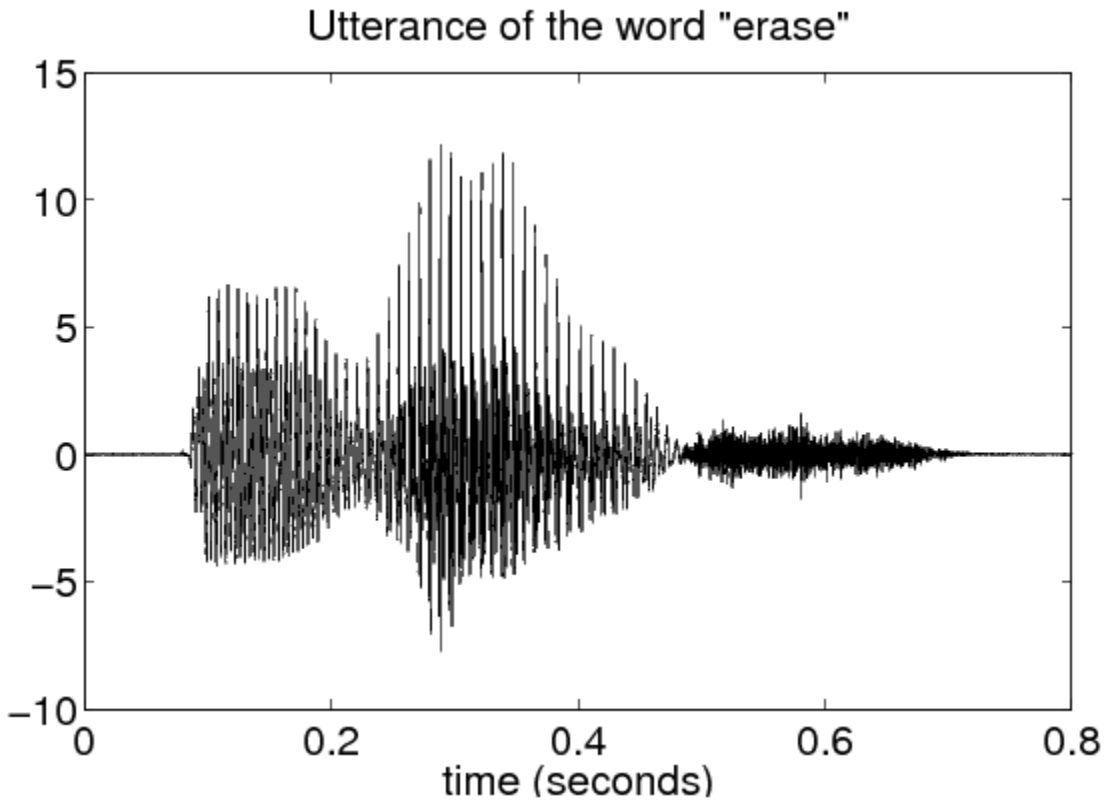
Voiced sounds, vowels for example, are produced by forcing air through the larynx, with the tension of the vocal cords adjusted so that they vibrate in a relaxed oscillation. This produces quasi-periodic pulses of air which are acoustically filtered as they propagate through the vocal tract, and possibly through the nasal cavity. The shape of the cavities that comprise the vocal tract, known as the **area function**, determines the natural frequencies, or **formants**, which are emphasized in the speech waveform. The period of the excitation, known as the **pitch period**, is generally small with respect to the rate at which the vocal tract changes shape. Therefore, a segment of voiced speech covering several pitch periods will appear somewhat **periodic**. Average values for the pitch period are around 8 ms for male speakers, and 4 ms for female speakers.

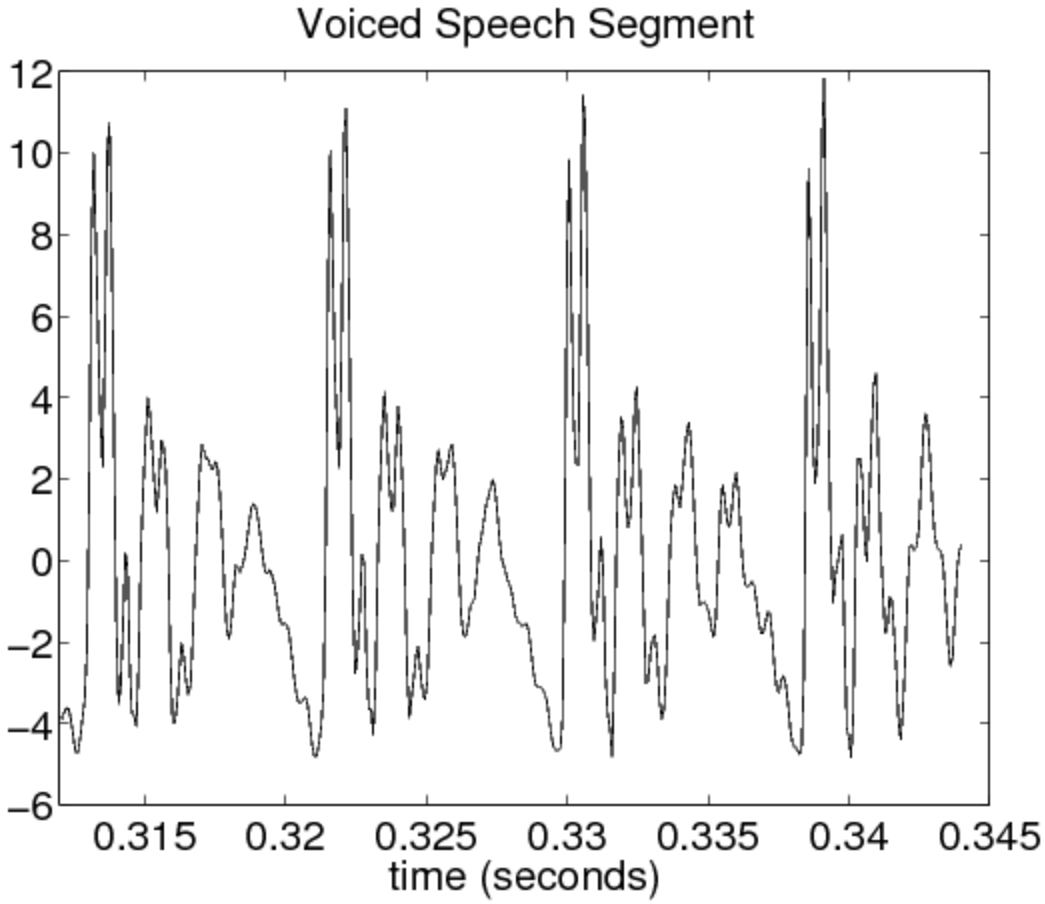
In contrast, unvoiced speech has more of a noise-like quality. Unvoiced sounds are usually much smaller in amplitude, and oscillate much faster than voiced speech. These sounds are generally produced by turbulence, as air is forced through a constriction at some point in the vocal tract. For example, an **h** sound comes from a constriction at the vocal cords, and an **f** is generated by a constriction at the lips.

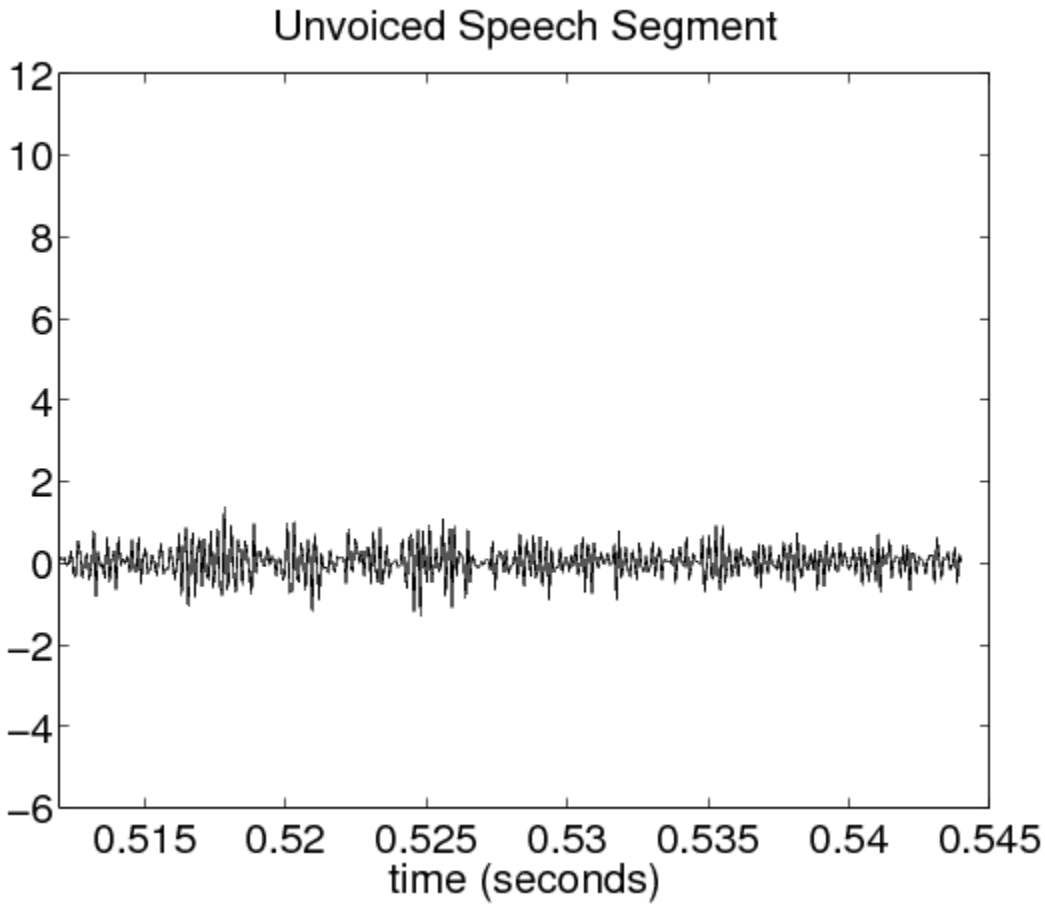
An illustrative example of voiced and unvoiced sounds contained in the word “erase” are shown in [link]. The original utterance is shown in (2.1). The voiced segment in (2.2) is a time magnification of the “a” portion of the word. Notice the highly periodic nature of this segment. The fundamental period of this waveform, which is about 8.5 ms here, is called the **pitch**

period. The unvoiced segment in (2.3) comes from the “s” sound at the end of the word. This waveform is much more noise-like than the voiced segment, and is much smaller in magnitude.

(2.1) Utterance of the word “erase”. (2.2) Voiced segment. (2.3) Unvoiced segment.




Voiced Speech Segment
12
10
8
6
4
2
0
−2
−4
−6
0.315
0.32
0.325
0.33
0.335
0.34
0.345
time (seconds)



Classification of Voiced/Unvoiced Speech

Download the file start.au for the following sections. Click here for help on how to load and play audio signals.

For many speech processing algorithms, a very important step is to determine the type of sound that is being uttered in a given time frame. In this section, we will introduce two simple methods for discriminating between voiced and unvoiced speech.

Download the file start.au, and use the `auread()` function to load it into the Matlab workspace.

Do the following:

- Plot (not stem) the speech signal. Identify two segments of the signal: one segment that is voiced and a second segment that is unvoiced (the `zoom xon` command is useful for this). Circle the regions of the plot corresponding to these two segments and label them as voiced or unvoiced.
- Save 300 samples from the voiced segment of the speech into a Matlab vector called **VoicedSig**.
- Save 300 samples from the unvoiced segment of the speech into a Matlab vector called **UnvoicedSig**.
- Use the `subplot()` command to plot the two signals, **VoicedSig** and **UnvoicedSig** on a single figure.

> **Note:** Hand in your labeled plots. Explain how you selected your voiced and unvoiced regions.

Estimate the pitch period for the voiced segment. Keep in mind that these speech signals are sampled at 8 KHz, which means that the time between samples is 0.125 milliseconds (ms). Typical values for the pitch period are 8 ms for male speakers, and 4 ms for female speakers. Based on this, would you predict that the speaker is male, or female?

One way to categorize speech segments is to compute the average energy, or power. Recall this is defined by the following:
Equation:

$$P = \frac{1}{L} \sum_{n=1}^{L} x^2 (n)$$

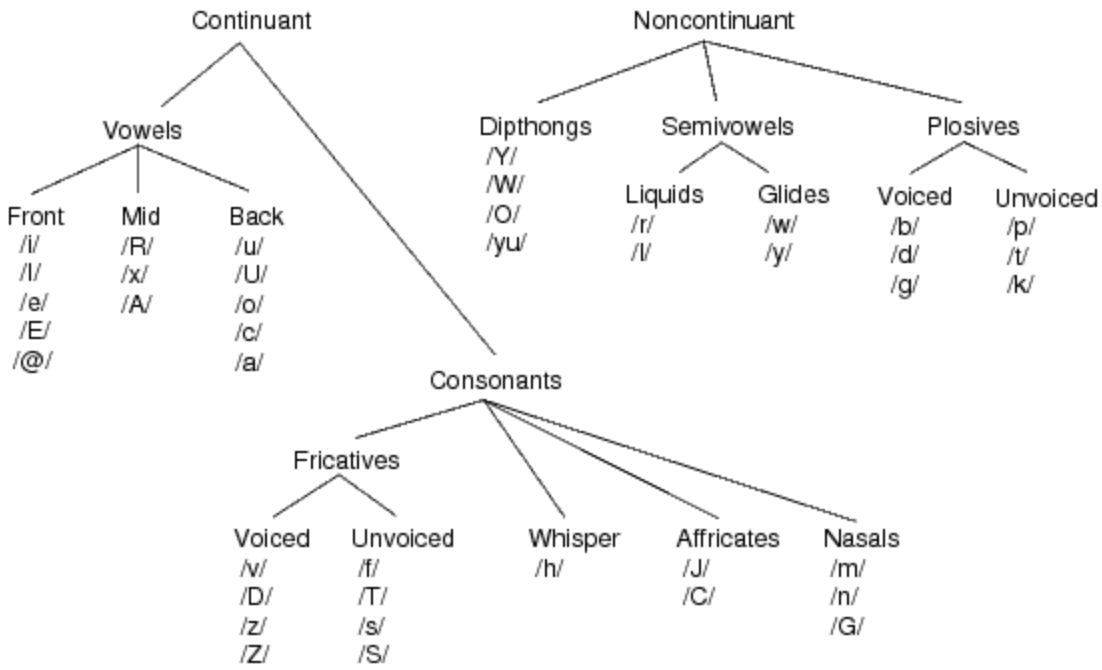
where L is the length of the frame $x(n)$. Use [link] to compute the average energy of the voiced and unvoiced segments that you plotted above. For which segment is the average energy greater?

Another method for discriminating between voiced and unvoiced segments is to determine the rate at which the waveform oscillates by counting number of zero-crossings that occur within a frame (the number of times the signal changes sign). Write a function **zero_cross** that will compute the number of zero-crossings that occur within a vector, and apply this to the two vectors **VoicedSig** and **UnvoicedSig**. Which segment has more zero-crossings?

INLAB REPORT

1. Give your estimate of the pitch period for the voiced segment, and your prediction of the gender of the speaker.
2. For each of the two vectors,**VoicedSig** and **UnvoicedSig**, list the average energy and number of zero-crossings. Which segment has a greater average energy? Which segment has a greater zero-crossing rate?
3. Hand in your **zero_cross** function.

Phonemes



Phonemes in American English. See [link] for more details.

American English can be described in terms of a set of about 42 distinct sounds called **phonemes**, illustrated in [link]. They can be classified in many ways according to their distinguishing properties. **Vowels** are formed by exciting a fixed vocal tract with quasi-periodic pulses of air. **Fricatives** are produced by forcing air through a constriction (usually towards the mouth end of the vocal tract), causing turbulent air flow. Fricatives may be voiced or unvoiced. **Plosive** sounds are created by making a complete closure, typically at the frontal vocal tract, building up pressure behind the closure and abruptly releasing it. A **diphthong** is a gliding monosyllabic sound that starts at or near the articulatory position for one vowel, and moves toward the position of another. It can be a very insightful exercise to recite the phonemes shown in [link], and make a note of the movements you are making to create them.

It is worth noting at this point that classifying speech sounds as voiced/unvoiced is not equivalent to the vowel/consonant distinction. Vowels and consonants are **letters**, whereas voiced and unvoiced refer to

types of speech **sounds**. There are several consonants, /m/ and /n/ for example, which when spoken are actually voiced sounds.

Short-Time Frequency Analysis

As we have seen from previous sections, the properties of speech signals are continuously changing, but may be considered to be stationary within an appropriate time frame. If analysis is performed on a "segment-by-segment" basis, useful information about the construction of an utterance may be obtained. The average energy and zero-crossing rate, as previously discussed, are examples of short-time feature extraction in the time-domain. In this section, we will learn how to obtain short-time frequency information from generally non-stationary signals.

stDTFT

Download the file go.au for the following section.

A useful tool for analyzing the spectral characteristics of a non-stationary signal is the **short-time discrete-time Fourier Transform**, or **stDTFT**, which we will define by the following:
Equation:

$$X_m\left(e^{j\omega}\right) = \sum_{n=-\infty}^{\infty} x\left(n\right) w\left(n-m\right) e^{-j\omega n}$$

Here, $x(n)$ is our speech signal, and $w(n)$ is a window of length L. Notice that if we fix m, the stDTFT is simply the DTFT of $x(n)$ multiplied by a shifted window. Therefore, $X_m\left(e^{j\omega}\right)$ is a collection of DTFTs of windowed segments of $x(n)$.

As we examined in the Digital Filter Design lab, windowing in the time domain causes an undesirable ringing in the frequency domain. This effect can be reduced by using some form of a raised cosine for the window $w(n)$.

Write a function `X = DFTwin(x,L,m,N)` that will compute the DFT of a windowed length L segment of the vector x.

- You should use a Hamming window of length L to window x.
- Your window should start at the index m of the signal x.
- Your DFTs should be of length N.
- You may use Matlab's `fft()` function to compute the DFTs.

Now we will test your `DFTwin()` function. Download the file go.au, and load it into Matlab. Plot the signal and locate a voiced region. Use your function to compute a 512-point DFT of a speech segment, with a window that covers six pitch periods within the voiced region. `Subplot` your chosen segment and the DFT magnitude (for ω from 0 to π) in the same figure. Label the frequency axis in Hz, assuming a sampling frequency of 8 KHz. Remember that a radial frequency of π corresponds to half the sampling frequency.

Note: Hand in the code for your

`DFTwin()`

function, and your plot. Describe the general shape of the spectrum, and estimate the formant frequencies for the region of voiced speech.

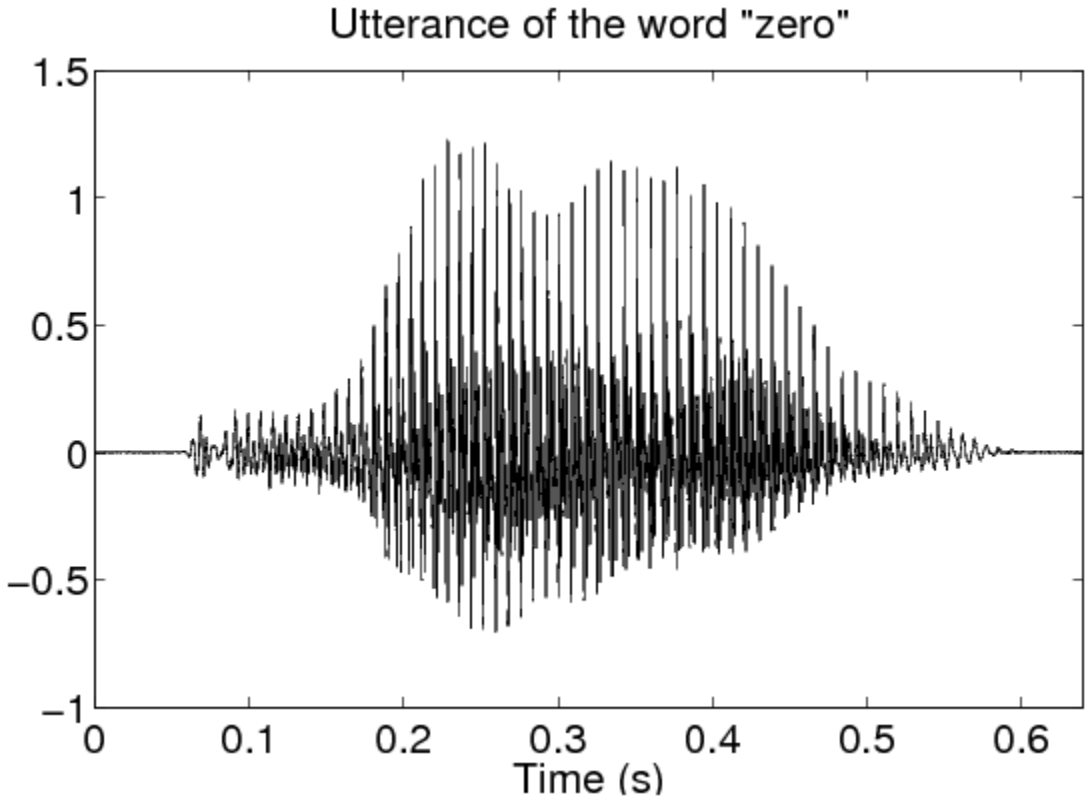
The Spectrogram

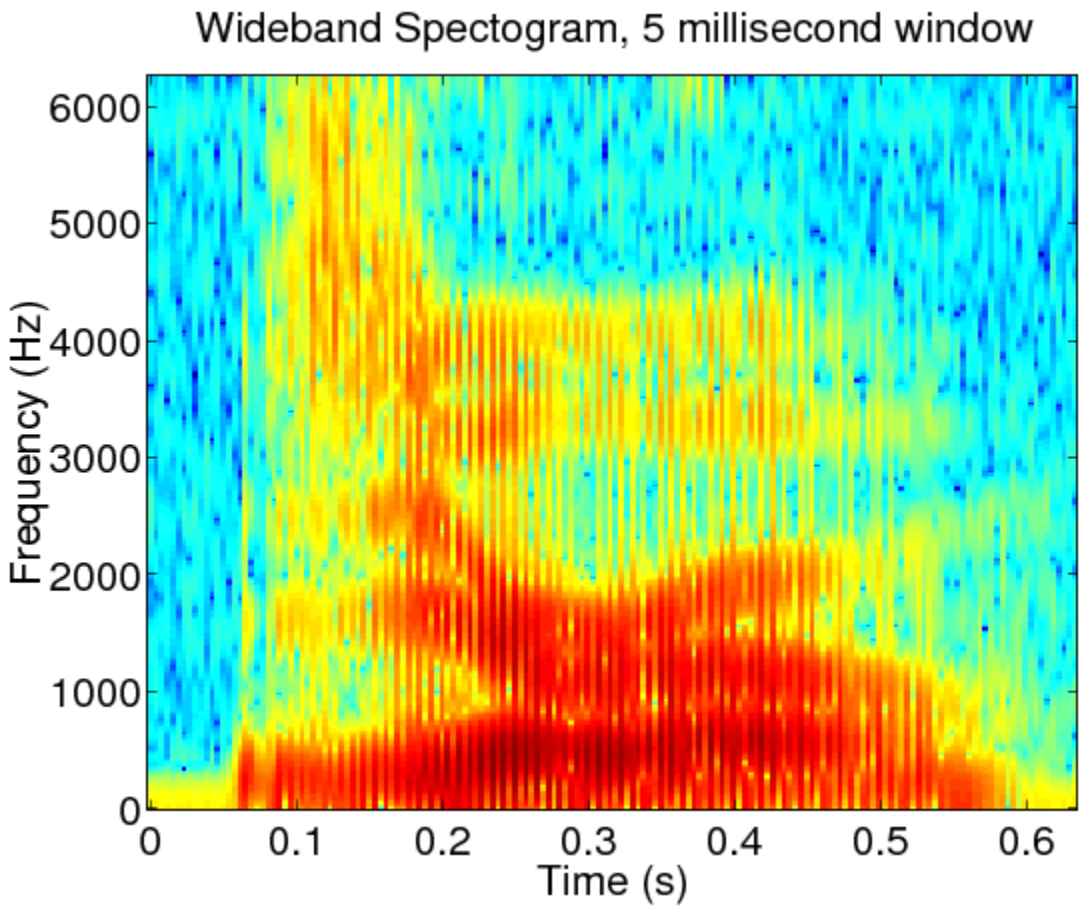
Download the file signal.mat for the following section.

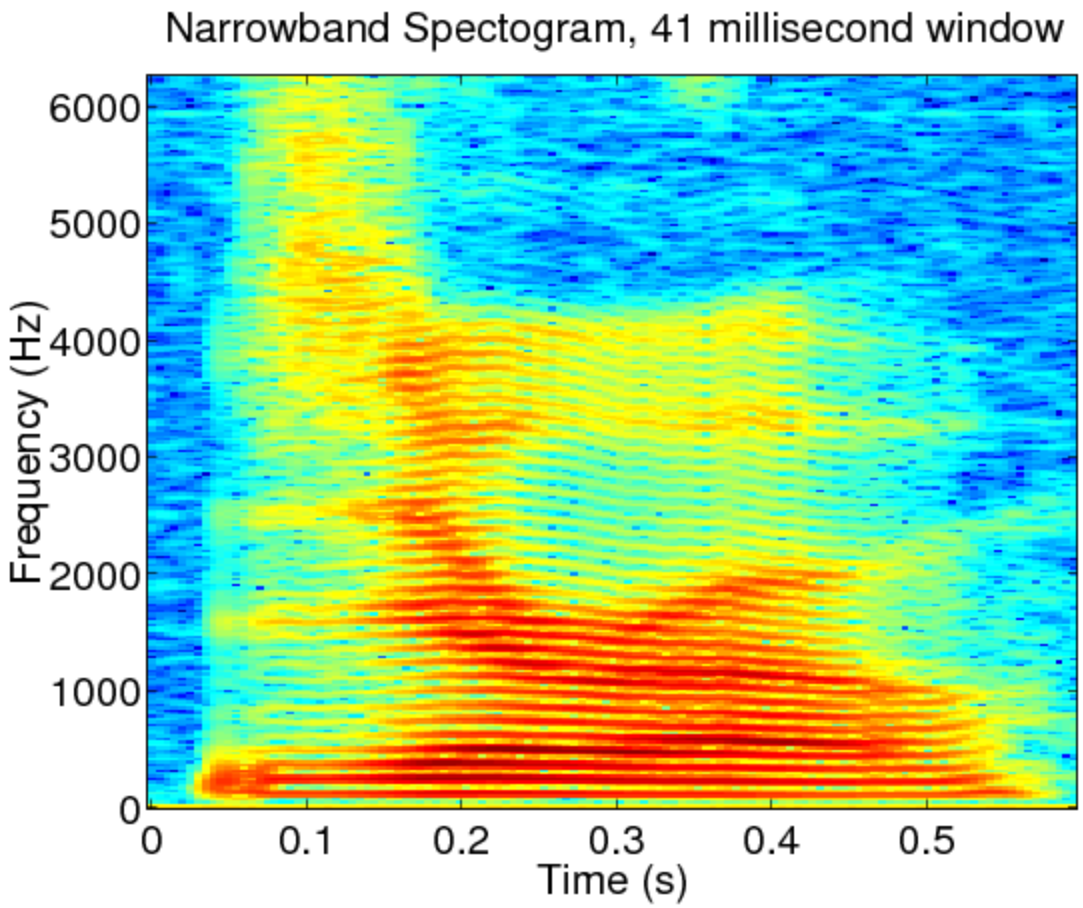
As previously stated, the short-time DTFT is a collection of DTFTs that differ by the position of the truncating window. This may be visualized as an image, called a **spectrogram**. A spectrogram shows how the spectral characteristics of the signal evolve with time. A spectrogram is created by placing the DTFTs vertically in an image, allocating a different column for each time segment. The convention is usually such that frequency increases

from bottom to top, and time increases from left to right. The pixel value at each point in the image is proportional to the magnitude (or squared magnitude) of the spectrum at a certain frequency at some point in time. A spectrogram may also use a “pseudo-color” mapping, which uses a variety of colors to indicate the magnitude of the frequency content, as shown in [link].

(1) Utterance of the word “zero”. (2) Wideband Spectrogram. (3) Narrowband Spectrogram.




Wideband Spectogram, 5 millisecond window
Frequency (Hz)
6000
5000
4000
3000
2000
1000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
Time (s)



For quasi-periodic signals like speech, spectrograms are placed into two categories according to the length of the truncating window. **Wideband** spectrograms use a window with a length comparable to a single period. This yields high resolution in the time domain but low resolution in the frequency domain. These are usually characterized by vertical striations, which correspond to high and low energy regions within a single period of the waveform. In **narrowband** spectrograms, the window is made long enough to capture several periods of the waveform. Here, the resolution in time is sacrificed to give a higher resolution of the spectral content. Harmonics of the fundamental frequency of the signal are resolved, and can be seen as horizontal striations. Care should be taken to keep the window short enough, such that the signal properties stay relatively constant within the window.

Often when computing spectrograms, not every possible window shift position is used from the stDTFT, as this would result in mostly redundant

information. Successive windows usually start many samples apart, which can be specified in terms of the **overlap** between successive windows. Criteria in deciding the amount of overlap include the length of the window, the desired resolution in time, and the rate at which the signal characteristics are changing with time.

Given this background, we would now like you to create a spectrogram using your `DFTwin()` function from the previous section. You will do this by creating a matrix of windowed DFTs, oriented as described above. Your function should be of the form

```
A = Specgm(x,L,overlap,N)
```

where `x` is your input signal, `L` is the window length, `overlap` is the number of points common to successive windows, and `N` is the number of points you compute in each DFT. Within your function, you should plot the magnitude (in dB) of your spectrogram matrix using the command `imagesc()`, and label the time and frequency axes appropriately.

Important Hints

- Remember that frequency in a spectrogram increases along the positive y-axis, which means that the first few elements of each column of the matrix will correspond to the highest frequencies.
- Your `DFTwin()` function returns the DT spectrum for frequencies between 0 and 2π. Therefore, you will only need to use the first or second half of these DFTs.
- The statement `B(:,n)` references the entire n^{th} column of the matrix B.
- In labeling the axes of the image, assume a sampling frequency of 8 KHz. Then the frequency will range from 0 to 4000 Hz.
- The `axis xy` command will be needed in order to place the origin of your plot in the lower left corner.
- You can get a standard gray-scale mapping (darker means greater magnitude) by using the command `colormap(1-gray)` , or a pseudo-color mapping using the command `colormap(jet)` . For more information, see the online help for the image command.

Download the file signal.mat, and load it into Matlab. This is a raised square wave that is modulated by a sinusoid. What would the spectrum of this signal look like? Create both a wideband and a narrowband spectrogram using your `Specgm()` function for the signal.

- For the wideband spectrogram, use a window length of 40 samples and an overlap of 20 samples.
- For the narrowband spectrogram, use a window length of 320 samples, and an overlap of 60 samples.

Subplot the wideband and narrowband spectrograms, and the original signal in the same figure.

Note: Hand in your code for

`Specgm()`

and your plots. Do you see vertical striations in the wideband spectrogram? Similarly, do you see horizontal striations in the narrowband spectrogram? In each case, what causes these lines, and what does the spacing between them represent?

Formant Analysis

Download the file vowels.mat for the following section.

The shape of an acoustic excitation for voiced speech is similar to a triangle wave. Therefore it has many harmonics at multiples of its fundamental frequency, $1/T_p$. As the excitation propagates through the vocal tract, acoustic resonances, or standing waves, cause certain harmonics to be significantly amplified. The specific wavelengths, hence the frequencies, of the resonances are determined by the shape of the cavities that comprise the vocal tract. Different vowel sounds are distinguished by unique sets of these

resonances, or **formant frequencies**. The first three average formants for several vowels are given in [link].

Formant Frequencies for the Vowels

Typewritten Symbol for the Vowel	Typical Word	F1 (Hz)	F2 (Hz)	F3 (Hz)
IY	(beet)	270	2290	3010
I	(bit)	390	1990	2550
E	(bet)	530	1840	2480
AE	(bat)	660	1720	2410
UH	(but)	520	1190	2390
A	(hot)	730	1090	2440
OW	(bought)	570	840	2410
U	(foot)	440	1020	2240
OO	(boot)	300	870	2240
ER	(bird)	490	1350	1690

Average Formant Frequencies for the Vowels.
See [link].

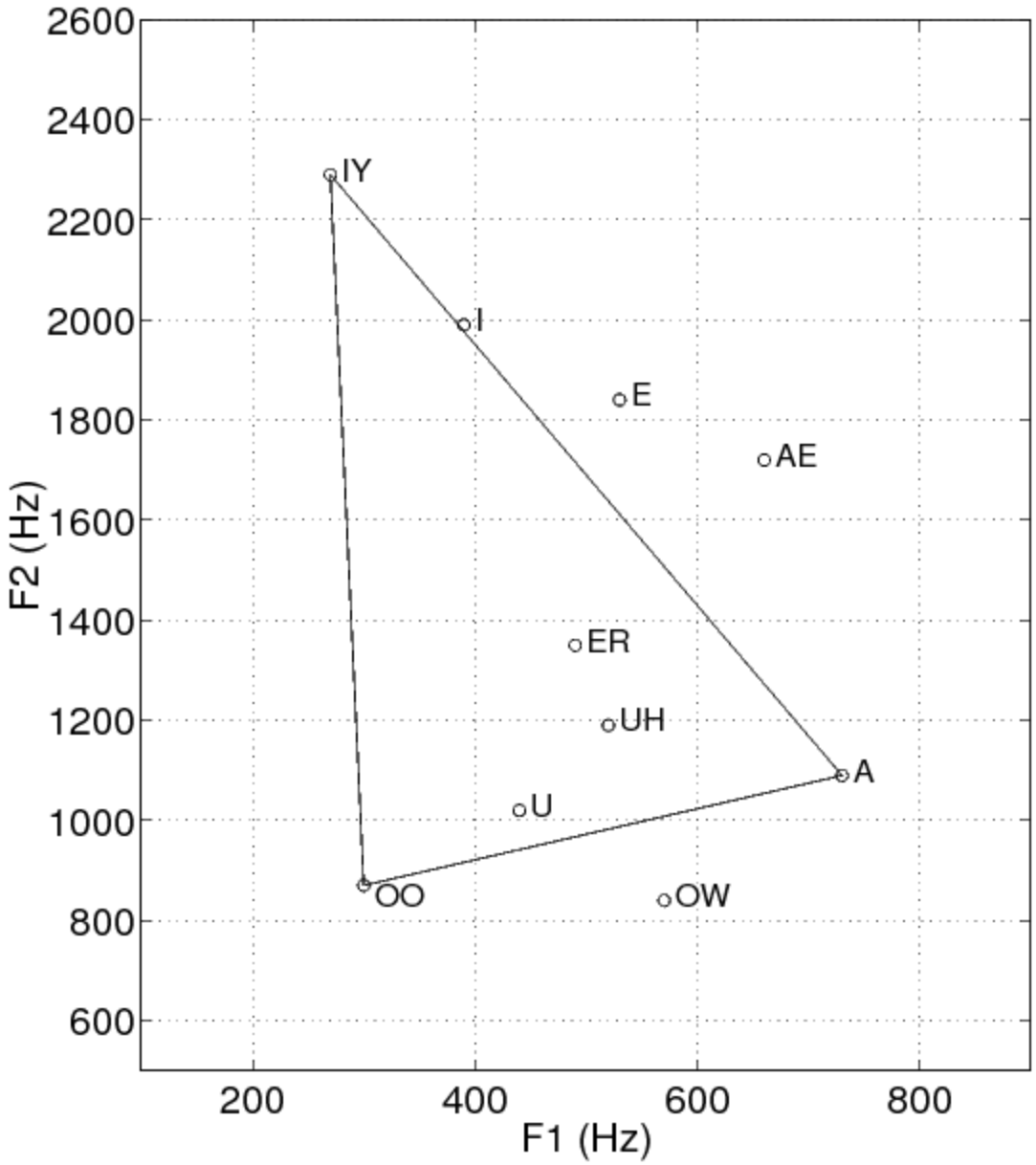
A possible technique for speech recognition would to determine a vowel utterance based on its unique set of formant frequencies. If we construct a graph that plots the second formant versus the first, we find that a particular vowel sound tends to lie within a certain region of the plane. Therefore, if we determine the first two formants, we can construct decision regions to estimate which vowel was spoken. The first two average formants for some common vowels are plotted in [link]. This diagram is known as the **vowel triangle** due to the general orientation of the average points.

Keep in mind that there is a continuous range of vowel sounds that can be produced by a speaker. When vowels are used in speech, their formants most often slide from one position to another.

Download the file vowels.mat, and load it into Matlab. This file contains the vowel utterances **a,e,i,o**, and **u** from a female speaker. Load this into Matlab, and plot a narrowband spectrogram of each of the utterances. Notice how the formant frequencies change with time.

For the vowels **a** and **u**, estimate the first two formant frequencies using the functions you created in the previous sections. Make your estimates at a time frame toward the beginning of the utterance, and another set of estimates toward the end of the utterance. You may want to use both the `Specgm` and `DFTwin` functions to determine the formants. Plot these four points in the vowel triangle provided in [link]. For each vowel, draw a line connecting the two points, and draw an arrow indicating the direction the formants are changing as the vowel is being uttered.

Note: Hand in your formant estimates on the vowel triangle.



The Vowel Triangle

Lab 9b - Speech Processing (part 2)

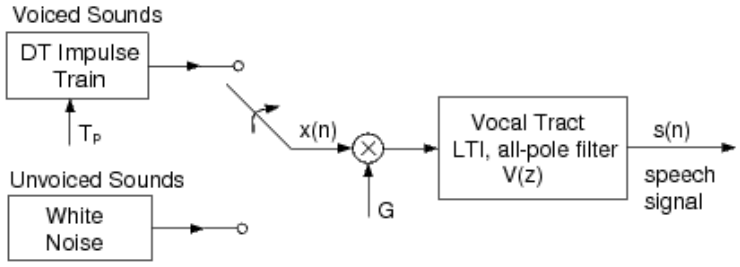
Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

This is the second part of a two week experiment. During the first week we discussed basic properties of speech signals, and performed some simple analyses in the time and frequency domain.

This week, we will introduce a system model for speech production. We will cover some background on **linear predictive coding**, and the final exercise will bring all the prior material together in a speech coding exercise.

A Speech Model



Discrete-Time Speech Production Model

From a signal processing standpoint, it is very useful to think of speech production in terms of a model, as in [link]. The model shown is the simplest of its kind, but it includes all the principal components. The excitations for voiced and unvoiced speech are represented by an impulse train and white noise generator, respectively. The pitch of voiced speech is controlled by the spacing between impulses, T_p, and the amplitude (volume) of the excitation is controlled by the gain factor G.

As the acoustical excitation travels from its source (vocal cords, or a constriction), the shape of the vocal tract alters the spectral content of the signal. The most

prominent effect is the formation of resonances, which intensifies the signal energy at certain frequencies (called **formants**). As we learned in the Digital Filter Design lab, the amplification of certain frequencies may be achieved with a linear filter by an appropriate placement of poles in the transfer function. This is why the filter in our speech model utilizes an all-pole LTI filter. A more accurate model might include a few zeros in the transfer function, but if the order of the filter is chosen appropriately, the all-pole model is sufficient. The primary reason for using the all-pole model is the distinct computational advantage in calculating the filter coefficients, as will be discussed shortly.

Recall that the transfer function of an all-pole filter has the form
Equation:

$$V(z) = \frac{1}{1 - \sum_{k=1}^{P} a_k z^{-k}}$$

where P is the order of the filter. This is an IIR filter that may be implemented with a recursive difference equation. With the input $G \cdot x(n)$, the speech signal $s(n)$ may be written as
Equation:

$$s(n) = \sum_{k=1}^{P} a_k s(n-k) + G \cdot x(n)$$

Keep in mind that the filter coefficients will change continuously as the shape of the vocal tract changes, but speech segments of an appropriately small length may be approximated by a time-invariant model.

This speech model is used in a variety of speech processing applications, including methods of speech recognition, speech coding for transmission, and speech synthesis. Each of these applications of the model involves dividing the speech signal into short segments, over which the filter coefficients are almost constant. For example, in speech transmission the bit rate can be significantly reduced by dividing the signal up into segments, computing and sending the model parameters for each segment (filter coefficients, gain, etc.), and re-synthesizing the signal at the receiving end, using a model similar to [link]. Most telephone systems use some form of this approach. Another example is speech recognition. Most recognition methods involve comparisons between short segments of the speech signals, and the filter coefficients of this model are often used in computing the “difference" between segments.

Synthesis of Voiced Speech

Download the file coeff.mat for the following section.

Download the file coeff.mat and load it into the Matlab workspace using the `load` command. This will load three sets of filter coefficients: $A1$, $A2$, and $A3$ for the vocal tract model in [link] and [link]. Each vector contains coefficients $\{a_1, a_2, \cdots, a_{15}\}$ for an all-pole filter of order 15.

We will now synthesize voiced speech segments for each of these sets of coefficients. First write a Matlab function `x=exciteV(N,Np)` which creates a length N excitation for voiced speech, with a pitch period of **Np** samples. The output vector x should contain a discrete-time impulse train with period **Np** (e.g. [1 0 0 $\cdots$ 0 1 0 0 $\cdots$]).

Assuming a sampling frequency of 8 kHz (0.125 ms/sample), create a 40 millisecond-long excitation with a pitch period of 8 ms, and filter it using [link] for each set of coefficients. For this, you may use the command

```
s = filter(1,[1 -A],x)
```

where A is the row vector of filter coefficients (see Matlab's help on **filter** for details). Plot each of the three filtered signals. Use `subplot()` and `orient tall` to place them in the same figure.

We will now compute the frequency response of each of these filters. The frequency response may be obtained by evaluating [link] at points along $z = e^{j\omega}$. Matlab will compute this with the command `[H,W]=freqz(1,[1 -A],512)` , where A is the vector of coefficients. Plot the magnitude of each response versus frequency in Hertz. Use `subplot()` and `orient tall` to plot them in the same figure.

The location of the peaks in the spectrum correspond to the formant frequencies. For each vowel signal, estimate the first three formants (in Hz) and list them in the figure.

Now generate the three signals again, but use an excitation which is 1-2 seconds long. Listen to the filtered signals using `soundsc`. Can you hear qualitative differences in the signals? Can you identify the vowel sounds?

INLAB REPORT
Hand in the following:

- A figure containing the three time-domain plots of the voiced signals.
- Plots of the frequency responses for the three filters. Make sure to label the frequency axis in units of Hertz.
- For each of the three filters, list the approximate center frequency of the first three formant peaks.
- Comment on the audio quality of the synthesized signals.

Linear Predictive Coding

The filter coefficients which were provided in the previous section were determined using a technique called **linear predictive coding** (LPC). LPC is a fundamental component of many speech processing applications, including compression, recognition, and synthesis.

In the following discussion of LPC, we will view the speech signal as a discrete-time random process.

Forward Linear Prediction

Suppose we have a discrete-time random process $\{\ldots, S_{-1}, S_0, S_1, S_2, \ldots\}$ whose elements have some degree of correlation. The goal of **forward linear prediction** is to predict the sample S_n using a linear combination of the previous P samples.
Equation:

$$\widehat{S}_n = \sum_{k=1}^{P} a_k S_{n-k}$$

P is called the **order** of the predictor. We may represent the error of predicting S_n by a random sequence e_n.
Equation:

$$\begin{aligned} e_n &= S_n - \widehat{S}_n \\ e_n &= S_n - \sum_{k=1}^{P} a_k S_{n-k} \end{aligned}$$

An optimal set of prediction coefficients a_k for [link] may be determined by minimizing the mean-square error $E\left[e_n^2\right]$. Note that since the error is generally a function of n, the prediction coefficients will also be functions of n. To simplify notation, let us first define the following column vectors.

Equation:

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 \cdots a_P \end{bmatrix}^T$$

Equation:

$$\mathbf{S}_{n,P} = \begin{bmatrix} S_{n-1} & S_{n-2} \cdots S_{n-P} \end{bmatrix}^T$$

Then,

Equation:

$$\begin{aligned} E\left[e_n^2\right] &= E\left[\left(S_n - \sum_{k=1}^{P} a_k S_{n-k}\right)^2\right] \\ &= E\left[\left(S_n - \mathbf{a}^T \mathbf{S}_{n,P}\right)^2\right] \\ &= E\left[S_n^2 - 2S_n \mathbf{a}^T \mathbf{S}_{n,P} + \mathbf{a}^T \mathbf{S}_{n,P}\ \mathbf{a}^T \mathbf{S}_{n,P}\right] \\ &= E\left[S_n^2\right] - 2\mathbf{a}^T E[S_n \mathbf{S}_{n,P}] + \mathbf{a}^T E\left[\mathbf{S}_{n,P} \mathbf{S}_{n,P}^T\right] \mathbf{a} \end{aligned}$$

The second and third terms of [link] may be written in terms of the autocorrelation sequence $r_{SS}\left(k,l\right)$.

Equation:

$$E\left[S_n \mathbf{S}_{n,P}\right] = \begin{bmatrix} E[S_n S_{n-1}] \\ E[S_n S_{n-2}] \\ \vdots \\ E[S_n S_{n-P}] \end{bmatrix} = \begin{bmatrix} r_{SS}\left(n, n-1\right) \\ r_{SS}\left(n, n-2\right) \\ \vdots \\ r_{SS}\left(n, n-P\right) \end{bmatrix} \equiv \mathbf{r}_S$$

Equation:

$$E\left[\mathbf{S}_{n,P}\mathbf{S}_{n,P}^T\right] = E\begin{bmatrix} S_{n-1}S_{n-1} & S_{n-1}S_{n-2} & \cdots & S_{n-1}S_{n-P} \\ S_{n-2}S_{n-1} & S_{n-2}S_{n-2} & \cdots & S_{n-2}S_{n-P} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n-P}S_{n-1} & S_{n-P}S_{n-2} & \cdots & S_{n-P}S_{n-P} \end{bmatrix}$$

$$= \begin{bmatrix} r_{SS}\left(n-1,n-1\right) & r_{SS}\left(n-1,n-2\right) & \cdots & r_{SS}\left(n-1,n-P\right) \\ r_{SS}\left(n-2,n-1\right) & r_{SS}\left(n-2,n-2\right) & \cdots & r_{SS}\left(n-2,n-P\right) \\ \vdots & \vdots & \ddots & \vdots \\ r_{SS}\left(n-P,n-1\right) & r_{SS}\left(n-P,n-2\right) & \cdots & r_{SS}\left(n-P,n-P\right) \end{bmatrix} \equiv \mathbf{R}_S$$

Substituting into [link], the mean-square error may be written as
Equation:

$$E\left[e_n^2\right] = E\left[S_n^2\right] - 2\mathbf{a}^T\mathbf{r}_S + \mathbf{a}^T\mathbf{R}_S\mathbf{a}$$

Note that while $\mathbf{a}$ and $\mathbf{r}_S$ are vectors, and $\mathbf{R}_S$ is a matrix, the expression in [link] is still a scalar quantity.

To find the optimal a_k coefficients, which we will call $\widehat{\mathbf{a}}$, we differentiate [link] with respect to the vector $\mathbf{a}$ (compute the gradient), and set it equal to the zero vector.
Equation:

$$\nabla_{\mathbf{a}} E\left[e_n^2\right] = -2\mathbf{r}_S + 2\mathbf{R}_S\widehat{\mathbf{a}} \;\equiv\; \mathbf{0}$$

Solving,
Equation:

$$\mathbf{R}_S\widehat{\mathbf{a}} = \mathbf{r}_S$$

The vector equation in [link] is a system of P scalar linear equations, which may be solved by inverting the matrix $\mathbf{R}_S$.

Note from [link] and [link] that $\mathbf{r}_S$ and $\mathbf{R}_S$ are generally functions of n. However, if S_n is wide-sense stationary, the autocorrelation function is only dependent on the difference between the two indices, $r_{SS}\left(k,l\right) = r_{SS}(|k-l|)$. Then $\mathbf{R}_S$ and $\mathbf{r}_S$ are no longer dependent on n, and may be written as follows.

Equation:

$$\mathbf{r}_S = \begin{bmatrix} r_{SS}(1) \\ r_{SS}(2) \\ \vdots \\ r_{SS}(P) \end{bmatrix}$$

Equation:

$$\mathbf{R}_S = \begin{bmatrix} r_{SS}(0) & r_{SS}(1) & \cdots & r_{SS}(P-1) \\ r_{SS}(1) & r_{SS}(0) & \cdots & r_{SS}(P-2) \\ r_{SS}(2) & r_{SS}(1) & \cdots & r_{SS}(P-3) \\ \vdots & \vdots & \ddots & \vdots \\ r_{SS}(P-1) & r_{SS}(P-2) & \cdots & r_{SS}(0) \end{bmatrix}$$

Therefore, if S_n is wide-sense stationary, the optimal a_k coefficients do not depend on n. In this case, it is also important to note that $\mathbf{R}_S$ is a Toeplitz (constant along diagonals) and symmetric matrix, which allows [link] to be solved efficiently using the Levinson-Durbin algorithm (see [link]). This property is essential for many real-time applications of linear prediction.

Linear Predictive Coding of Speech

An important question has yet to be addressed. The solution in [link] to the linear prediction problem depends entirely on the autocorrelation sequence. How do we estimate the autocorrelation of a speech signal? Recall that the applications to which we are applying LPC involve dividing the speech signal up into short segments and computing the filter coefficients for each segment. Therefore we need to consider the problem of estimating the autocorrelation for a short segment of the signal. In LPC, the following "biased" autocorrelation estimate is often used.
Equation:

$$\hat{r}_{SS}(m) = \frac{1}{N} \sum_{n=0}^{N-m-1} s(n) s(n+m), \quad 0 \leq m \leq P$$

Here we are assuming we have a length N segment which starts at $n = 0$. Note that this is the single-parameter form of the autocorrelation sequence, so that the forms in [link] and [link] may be used for $\mathbf{r}_S$ and $\mathbf{R}_S$.

LPC Exercise

Download the file test.mat for this exercise.

Write a function `coef=mylpc(x,P)` which will compute the order-P LPC coefficients for the column vector x, using the autocorrelation method ("lpc" is a built-in Matlab function, so use the name **mylpc**). Consider the input vector x as a speech segment, in other words do not divide it up into pieces. The output vector **coef** should be a column vector containing the P coefficients $\{\widehat{a}_1, \widehat{a}_2, \cdots, \widehat{a}_P\}$. In your function you should do the following:

1. Compute the biased autocorrelation estimate of [link] for the lag values $0 \leq m \leq P$. You may use the `xcorr` function for this.
2. Form the $\mathbf{r}_S$ and $\mathbf{R}_S$ vectors as in [link] and [link]. Hint: Use the `toeplitz` function to form $\mathbf{R}_S$.
3. Solve the matrix equation in [link] for $\widehat{\mathbf{a}}$.

To test your function, download the file test.mat, and **load** it into Matlab. This file contains two vectors: a signal x and its order-15 LPC coefficients a. Use your function to compute the order-15 LPC coefficients of x, and compare the result to the vector a.

Note: Hand in your

`mylpc`

function.

Speech Coding and Synthesis

Download the file phrase.au for the following section.

One very effective application of LPC is the compression of speech signals. For example, an **LPC vocoder** (voice-coder) is a system used in many telephone systems to reduce the bit rate for the transmission of speech. This system has two overall components: an analysis section which computes signal parameters (gain, filter coefficients, etc.), and a synthesis section which reconstructs the speech signal after transmission.

Since we have introduced the speech model in "A Speech Model", and the estimation of LPC coefficients in "Linear Predictive Coding", we now have all the tools necessary to implement a simple vocoder. First, in the analysis section, the original speech signal will be split into short time frames. For each frame, we will compute the signal energy, the LPC coefficients, and determine whether the segment is voiced or unvoiced.

Download the file phrase.au. This speech signal is sampled at a rate of 8000 Hz.

1. Divide the original speech signal into 30ms non-overlapping frames. Place the frames into L consecutive columns of a matrix S (use **reshape**). If the samples at the tail end of the signal do not fill an entire column, you may disregard these samples.
2. Compute the energy of each frame of the original word, and place these values in a length L vector called **energy**.
3. Determine whether each frame is voiced or unvoiced. Use your **zero_cross** function from the first week to compute the number of zero-crossings in each frame. For length N segments with less than $\frac{N}{2}$ zero-crossings, classify the segment as voiced, otherwise unvoiced. Save the results in a vector **VU** which takes the value of “1" for voiced and “0" for unvoiced.
4. Use your `mylpc` function to compute order-15 LPC coefficients for each frame. Place each set of coefficients into a column of a $15 \times L$ matrix A.

To see the reduction in data, add up the total number of bytes Matlab uses to store the encoded speech in the arrays **A**, **VU**, and **energy**. (use the `whos` function). Compute the **compression ratio** by dividing this by the number of bytes Matlab uses to store the original speech signal. Note that the compression ratio can be further improved by using a technique called **vector quantization** on the LPC coefficients, and also by using fewer bits to represent the gain and voiced/unvoiced indicator.

Now the computed parameters will be used to re-synthesize the phrase using the model in [link]. Similar to your `exciteV` function from "Synthesis of Voiced Speech", create a function `x=exciteUV(N)` which returns a length N excitation for unvoiced speech (generate a Normal(0,1) sequence). Then for each encoded frame do the following:

1. Check if current frame is voiced or unvoiced.
2. Generate the frame of speech by using the appropriate excitation into the filter specified by the LPC coefficients (you did this in "Synthesis of Voiced Speech"). For voiced speech, use a pitch period of 7.5 ms. Make sure your synthesized segment is the same length as the original frame.
3. Scale the amplitude of the segment so that the synthesized segment has the same energy as the original.
4. Append the frame to the end of the output vector.

Listen to the original and synthesized phrase. Can you recognize the synthesized version as coming from the same speaker? What are some possible ways to improve the quality of the synthesized speech? `Subplot` the two speech signals in the same figure.

INLAB REPORT
Hand in the following:

- Your analysis and synthesis code.
- The compression ratio.
- Plots of the original and synthesized words.
- Comment on the quality of your synthesized signal. How might the quality be improved?

Lab 10a - Image Processing (part 1)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

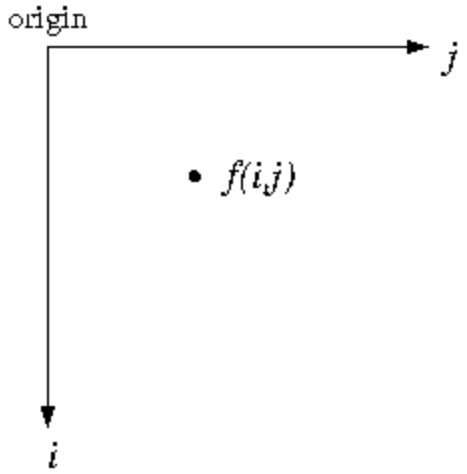
Introduction

This is the first part of a two week experiment in image processing. During this week, we will cover the fundamentals of digital monochrome images, intensity histograms, pointwise transformations, gamma correction, and image enhancement based on filtering.

In the second week , we will cover some fundamental concepts of color images. This will include a brief description on how humans perceive color, followed by descriptions of two standard color spaces. The second week will also discuss an application known as image **halftoning**.

Introduction to Monochrome Images

An **image** is the optical representation of objects illuminated by a light source. Since we want to process images using a computer, we represent them as functions of discrete spatial variables. For **monochrome** (black-and-white) images, a scalar function $f(i,j)$ can be used to represent the light **intensity** at each spatial coordinate (i,j). [link] illustrates the convention we will use for spatial coordinates to represent images.



Spatial coordinates used in digital image representation.

If we assume the coordinates to be a set of positive integers, for example $i = 1, \cdots, M$ and $j = 1, \cdots, N$, then an image can be conveniently represented by a matrix.
Equation:

$$f(i,j) = \begin{matrix} f(1,1) & f(1,2) & \cdots & f(1,N) \\ f(2,1) & f(2,2) & \cdots & f(2,N) \\ \vdots & \vdots & & \vdots \\ f(M,1) & f(M,2) & \cdots & f(M,N) \end{matrix}$$

We call this an $M \times N$ image, and the elements of the matrix are known as **pixels**.

The pixels in digital images usually take on integer values in the finite range,
Equation:

$$0 \leq f(i,j) \leq L_{max}$$

where 0 represents the minimum intensity level (black), and L_{max} is the maximum intensity level (white) that the digital image can take on. The interval $[0, L_{max}]$ is known as a **gray scale**.

In this lab, we will concentrate on 8-bit images, meaning that each pixel is represented by a single byte. Since a byte can take on 256 distinct values, L_{max} is 255 for an 8-bit image.

Exercise

Download the file yacht.tif for the following section. Click here for help on the Matlab image command.

In order to process images within Matlab, we need to first understand their numerical representation. Download the image file yacht.tif . This is an 8-bit monochrome image. Read it into a matrix using

```
A = imread('yacht.tif');
```

Type `whos` to display your variables. Notice under the "Class" column that the A matrix elements are of type **uint8** (unsigned integer, 8 bits). This means that Matlab is using a single byte to represent each pixel. Matlab cannot perform numerical computation on numbers of type **uint8**, so we usually need to convert the matrix to a floating point representation. Create a double precision representation of the image using `B = double(A);` . Again, type `whos` and notice the difference in the number of bytes between A and B. In future sections, we will be performing computations on our images, so we need to remember to convert them to type **double** before processing them.

Display **yacht.tif** using the following sequence of commands:

```
image(B);
```

```
colormap(gray(256));
```

```
axis('image');
```

The `image` command works for both type **uint8** and **double** images. The `colormap` command specifies the range of displayed gray levels, assigning black to 0 and white to 255. It is important to note that if any pixel values are outside the range 0 to 255 (after processing), they will be clipped to 0 or 255 respectively in the displayed image. It is also important to note that a floating point pixel value will be rounded down ("floored") to an integer before it is displayed. Therefore the maximum number of gray levels that will be displayed on the monitor is 255, even if the image values take on a continuous range.

Now we will practice some simple operations on the **yacht.tif** image. Make a horizontally flipped version of the image by reversing the order of each column. Similarly, create a vertically flipped image. Print your results.

Now, create a "negative" of the image by subtracting each pixel from 255 (here's an example of where conversion to **double** is necessary.) Print the result.

Finally, multiply each pixel of the original image by 1.5, and print the result.

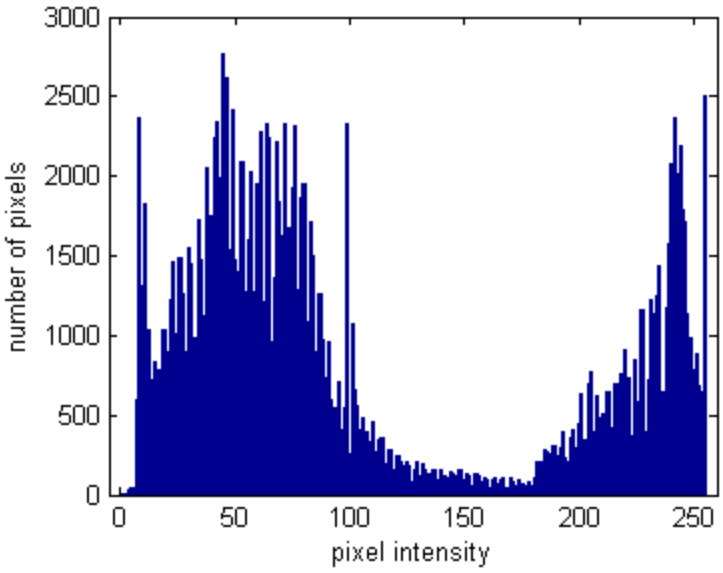
INLAB REPORT

1. Hand in two flipped images.
2. Hand in the negative image.
3. Hand in the image multiplied by factor of 1.5. What effect did this have?

Pixel Distributions

Download the files house.tif and narrow.tif for the following sections.

Histogram of an Image



Histogram of an 8-bit image

The **histogram** of a digital image shows how its pixel intensities are distributed. The pixel intensities vary along the horizontal axis, and the number of pixels at each intensity is plotted vertically, usually as a bar graph. A typical histogram of an 8-bit image is shown in [link].

Write a simple Matlab function `Hist(A)` which will plot the histogram of image matrix A. You may use Matlab's `hist` function, however that function requires a vector as input. An example of using `hist` to plot a histogram of a matrix would be

```
x=reshape(A,1,M*N);
```

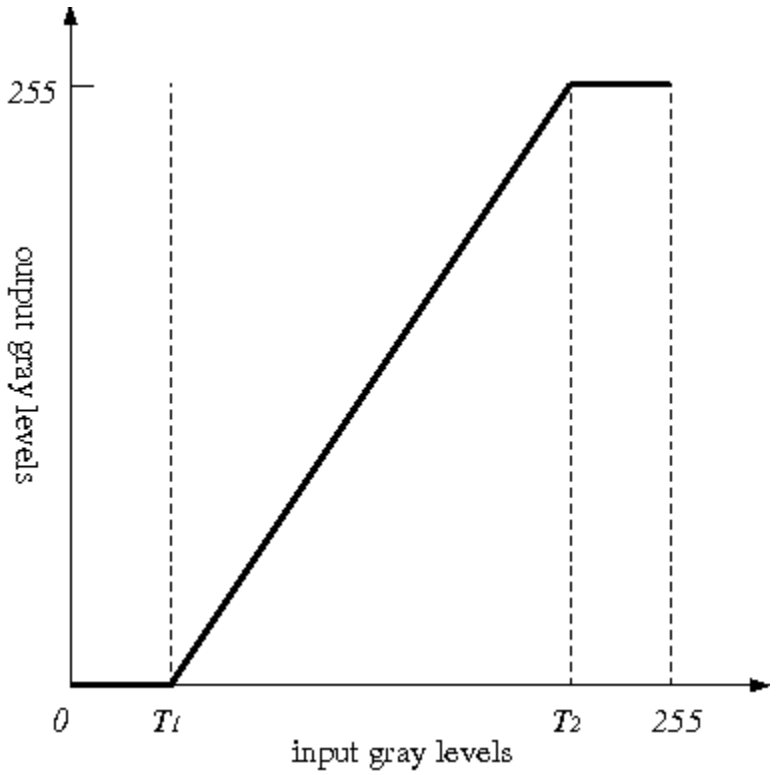
```
hist(x,0:255);
```

where A is an image, and M and N are the number of rows and columns in A. The `reshape` command is creating a row vector out of the image matrix, and the `hist` command plots a histogram with bins centered at $[0:255]$.

Download the image file house.tif , and read it into Matlab. Test your `Hist` function on the image. Label the axes of the histogram and give it a title.

Note: Hand in your labeled histogram. Comment on the distribution of the pixel intensities.

Pointwise Transformations



Pointwise transformation of image

A pointwise transformation is a function that maps pixels from one intensity to another. An example is shown in [link]. The horizontal axis shows all

possible intensities of the original image, and the vertical axis shows the intensities of the transformed image. This particular transformation maps the "darker" pixels in the range $[0, T_1]$ to a level of zero (black), and similarly maps the "lighter" pixels in $[T_2, 255]$ to white. Then the pixels in the range $[T_1, T_2]$ are "stretched out" to use the full scale of $[0, 255]$. This can have the effect of increasing the contrast in an image.

Pointwise transformations will obviously affect the pixel distribution, hence they will change the shape of the histogram. If a pixel transformation can be described by a one-to-one function, $y = f(x)$, then it can be shown that the input and output histograms are approximately related by the following:
Equation:

$$H_{out}(y) \approx \left. \frac{H_{in}(x)}{|f'(x)|} \right|_{x=f^{-1}(y)}.$$

Since x and y need to be integers in [link], the evaluation of $x = f^{-1}(y)$ needs to be rounded to the nearest integer.

The pixel transformation shown in [link] is not a one-to-one function. However, [link] still may be used to give insight into the effect of the transformation. Since the regions $[0, T_1]$ and $[T_2, 255]$ map to the single points 0 and 255, we might expect "spikes" at the points 0 and 255 in the output histogram. The region $[1, 254]$ of the output histogram will be directly related to the input histogram through [link].

First, notice from $x = f^{-1}(y)$ that the region $[1, 254]$ of the output is being mapped from the region $[T_1, T_2]$ of the input. Then notice that $f'(x)$ will be a constant scaling factor throughout the entire region of interest. Therefore, the output histogram should approximately be a stretched and rescaled version of the input histogram, with possible spikes at the endpoints.

Write a Matlab function that will perform the pixel transformation shown in [link]. It should have the syntax

`output = pointTrans(input, T1, T2)`.

Hints

- Determine an equation for the graph in [link], and use this in your function. Notice you have three input regions to consider. You may want to create a separate function to apply this equation.
- If your function performs the transformation one pixel at a time, be sure to allocate the space for the output image at the beginning to speed things up.

Download the image file narrow.tif and read it into Matlab. Display the image, and compute its histogram. The reason the image appears "washed out" is that it has a narrow histogram. Print out this picture and its histogram.

Now use your `pointTrans` function to spread out the histogram using $T1 = 70$ and $T2 = 180$. Display the new image and its histogram. (You can open another figure window using the `figure` command.) Do you notice a difference in the "quality" of the picture?

INLAB REPORT

1. Hand in your code for `pointTrans`.
2. Hand in the original image and its histogram.
3. Hand in the transformed image and its histogram.
4. What qualitative effect did the transformation have on the original image? Do you observe any negative effects of the transformation?
5. Compare the histograms of the original and transformed images. Why are there zeros in the output histogram?

Gamma Correction

Download the file dark.tif for the following section.

The light intensity generated by a physical device is usually a nonlinear function of the original signal. For example, a pixel that has a gray level of

200 will not be twice as bright as a pixel with a level of 100. Almost all computer monitors have a **power law** response to their applied voltage. For a typical cathode ray tube (CRT), the brightness of the illuminated phosphors is approximately equal to the applied voltage raised to a power of 2.5. The numerical value of this exponent is known as the **gamma** (γ) of the CRT. Therefore the power law is expressed as

Equation:

$$I = V^{\gamma}$$

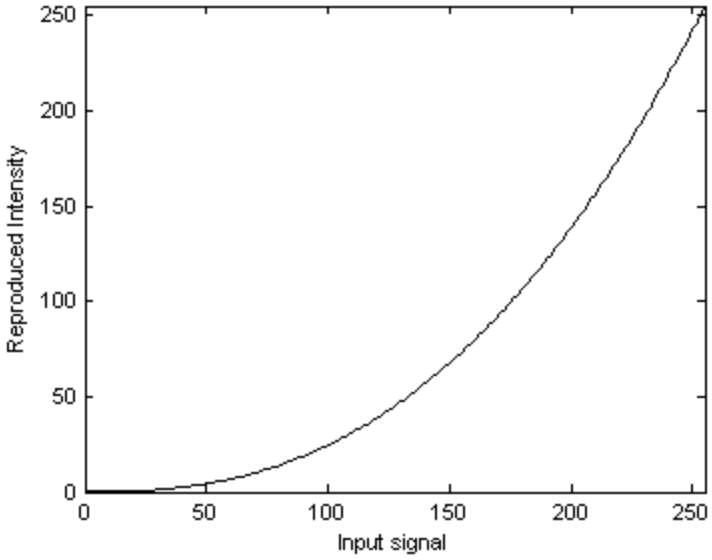
where I is the pixel intensity and V is the voltage applied to the device.

If we relate [link] to the pixel values for an 8-bit image, we get the following relationship,

Equation:

$$y = 255\left(\frac{x}{255}\right)^{\gamma}$$

where x is the original pixel value, and y is the pixel intensity as it appears on the display. This relationship is illustrated in [link].



Nonlinear behavior of a display device having a γ of 2.2.

In order to achieve the correct reproduction of intensity, this nonlinearity must be compensated by a process known as γ**correction**. Images that are not properly corrected usually appear too light or too dark. If the value of γ is available, then the correction process consists of applying the inverse of [link]. This is a straightforward pixel transformation, as we discussed in the section "Pointwise Transformations".

Write a Matlab function that will γ correct an image by applying the inverse of [link]. The syntax should be

```
B = gammCorr(A,gamma)
```

where A is the uncorrected image, $gamma$ is the γ of the device, and B is the corrected image. (See the hints in "Pointwise Transformations".)

The file dark.tif is an image that has not been γ corrected for your monitor. Download this image, and read it into Matlab. Display it and observe the quality of the image.

Assume that the γ for your monitor is 2.2. Use your `gammCorr` function to correct the image for your monitor, and display the resultant image. Did it improve the quality of the picture?

INLAB REPORT

1. Hand in your code for `gammCorr`.
2. Hand in the γ corrected image.
3. How did the correction affect the image? Does this appear to be the correct value for γ ?

Image Enhancement Based on Filtering

Sometimes, we need to process images to improve their appearance. In this section, we will discuss two fundamental image enhancement techniques: image **smoothing** and **sharpening**.

Image Smoothing

Smoothing operations are used primarily for diminishing spurious effects that may be present in a digital image, possibly as a result of a poor sampling system or a noisy transmission channel. Lowpass filtering is a popular technique of image smoothing.

Some filters can be represented as a 2-D convolution of an image $f(i,j)$ with the filter's impulse response $h(i,j)$.
Equation:

$$\begin{aligned} g(i,j) &= f(i,j)^{**}\ h(i,j) \\ &= \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f\,(k,l) h\,(i-k, j-l) \end{aligned}$$

Some typical lowpass filter impulse responses are shown in [link], where the center element corresponds to $h(0,0)$. Notice that the terms of each

filter sum to one. This prevents amplification of the DC component of the original image. The frequency response of each of these filters is shown in [link].

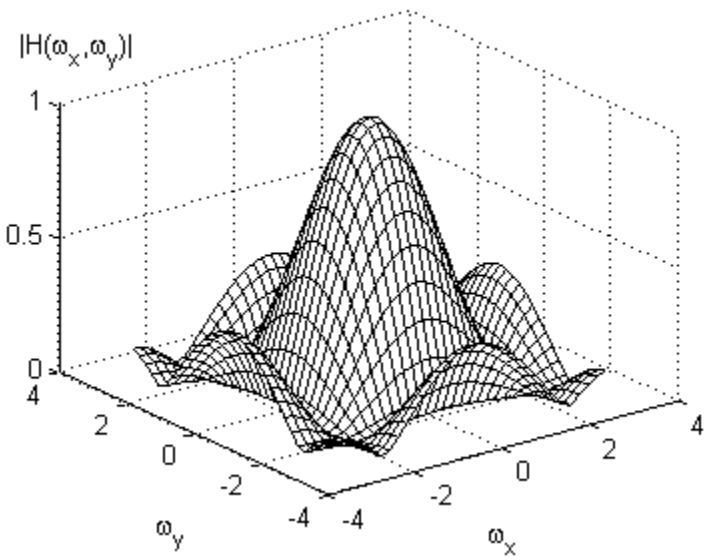
Impulse responses of lowpass filters useful for image smoothing.

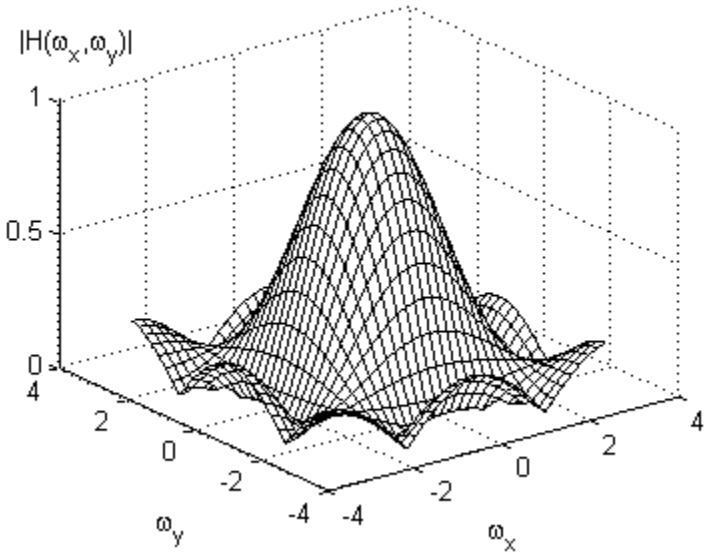
$$\frac{1}{9}\cdot\begin{array}{|c|c|c|}\hline 1&1&1\\\hline 1&1&1\\\hline 1&1&1\\\hline\end{array}$$

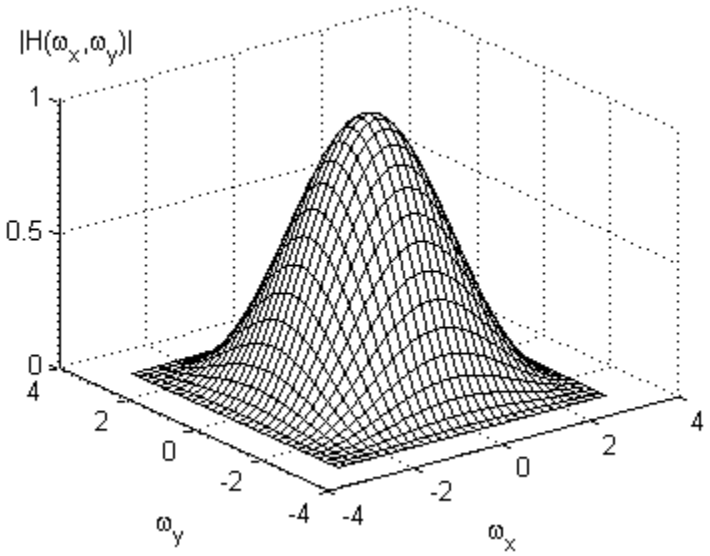
$$\frac{1}{10}\cdot\begin{array}{|c|c|c|}\hline 1&1&1\\\hline 1&2&1\\\hline 1&1&1\\\hline\end{array}$$

$$\frac{1}{16}\cdot\begin{array}{|c|c|c|}\hline 1&2&1\\\hline 2&4&2\\\hline 1&2&1\\\hline\end{array}$$

Frequency responses of the lowpass filters shown in Fig. 5.







An example of image smoothing is shown in [link], where the degraded image is processed by the filter shown in [link]. It can be seen that lowpass filtering clearly reduces the additive noise, but at the same time it **blurs** the image. Hence, blurring is a major limitation of lowpass filtering.

(1) Original gray scale image. (2) Original image degraded by additive white Gaussian noise, $N(0, 0.01)$. (3) Result of processing the

degraded image with a lowpass filter.

In addition to the above linear filtering techniques, images can be smoothed by **nonlinear** filtering, such as mathematical morphological processing. **Median** filtering is one of the simplest morphological techniques, and is useful in the reduction of impulsive noise. The main advantage of this type of filter is that it can reduce noise while preserving the detail of the original image. In a median filter, each input pixel is replaced by the median of the pixels contained in a surrounding window. This can be expressed by **Equation:**

$$g(i,j) = median\{f(i-k, j-l)\}, \ \ (k,l) \in W$$

where W is a suitably chosen window. [link] shows the performance of the median filter in reducing so-called "salt and pepper" noise.

(1) Original gray scale image. (2) Original image degraded by "salt and pepper" noise with 0.05 noise density. (3) Result of 3×3 median

filtering.

Smoothing Exercise

Download the files race.tif, noise1.tif and noise2.tif for this exercise. Click here for help on the Matlab mesh command.

Among the many spatial lowpass filters, the Gaussian filter is of particular importance. This is because it results in very good spatial and spectral localization characteristics. The Gaussian filter has the form
Equation:

$$h\left(i,j\right) = C \ \ \exp\left(-\frac{i^2+j^2}{2\sigma^2}\right)$$

where σ^2, known as the **variance**, determines the size of passband area. Usually the Gaussian filter is normalized by a scaling constant C such that the sum of the filter coefficient magnitudes is one, allowing the average intensity of the image to be preserved.

Equation:

$$\sum_{i,j} h\left(i,j\right) = 1$$

Write a Matlab function that will create a **normalized** Gaussian filter that is centered around the origin (the center element of your matrix should be $h(0,0)$). Note that this filter is both **separable** and **symmetric**, meaning $h(i,j) = h(i)h(j)$ and $h(i) = h(-i)$. Use the syntax

```
h=gaussFilter(N, var)
```

where `N` determines the size of filter, `var` is the variance, and `h` is the $N \times N$ filter. Notice that for this filter to be symmetrically centered around zero, N will need to be an odd number.

Use Matlab to compute the frequency response of a 7×7 Gaussian filter with $\sigma^2 = 1$. Use the command

```
H = fftshift(fft2(h,32,32));
```

to get a 32×32 DFT. Plot the magnitude of the frequency response of the Gaussian filter, $|H_{Gauss}\left(\omega_1, \omega_2\right)|$, using the `mesh` command. Plot it over the region $[-\pi, \pi] \times [-\pi, \pi]$, and label the axes.

Filter the image contained in the file race.tif with a 7×7 Gaussian filter, with $\sigma^2 = 1$.

Note: You can filter the signal by using the Matlab command `Y=filter2(h,X);` , where X is the matrix containing the input image and h is the impulse response of the filter.

Display the original and the filtered images, and notice the blurring that the filter has caused.

Now write a Matlab function to implement a 3×3 median filter (without using the `medfilt2` command). Use the syntax

```
Y = medianFilter(X);
```

where X and Y are the input and output image matrices, respectively. For convenience, you do not have to alter the pixels on the border of X.

Note: Use the Matlab command

`median`

to find the median value of a subarea of the image, i.e. a 3×3 window surrounding each pixel.

Download the image files noise1.tif and noise2.tif . These images are versions of the previous **race.tif** image that have been degraded by additive white Gaussian noise and "salt and pepper" noise, respectively. Read them into Matlab, and display them using `image`. Filter each of the noisy images with both the 7×7 Gaussian filter ($\sigma^2 = 1$) and the 3×3 median filter. Display the results of the filtering, and place a title on each figure. (You can open several figure windows using the `figure` command.) Compare the filtered images with the original noisy images. Print out the four filtered pictures.

INLAB REPORT

1. Hand in your code for `gaussFilter` and `medianFilter`.
2. Hand in the plot of $|H_{Gauss}(\omega_1, \omega_2)|$.
3. Hand in the results of filtering the noisy images (4 pictures).

4. Discuss the effectiveness of each filter for the case of additive white Gaussian noise. Discuss both positive and negative effects that you observe for each filter.
5. Discuss the effectiveness of each filter for the case of "salt and pepper" noise. Again, discuss both positive and negative effects that you observe for each filter.

Image Sharpening

Image sharpening techniques are used primarily to enhance an image by highlighting details. Since fine details of an image are the main contributors to its high frequency content, highpass filtering often increases the local contrast and sharpens the image. Some typical highpass filter impulse responses used for contrast enhancement are shown in [link]. The frequency response of each of these filters is shown in [link].

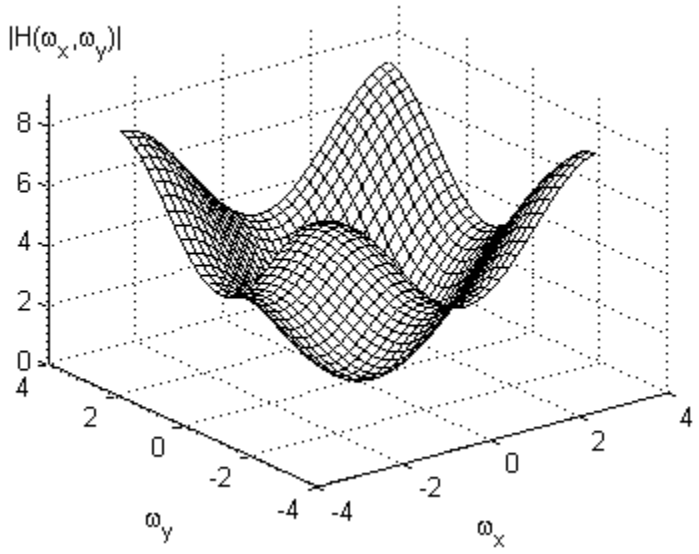
Impulse responses of highpass filters useful for image sharpening.

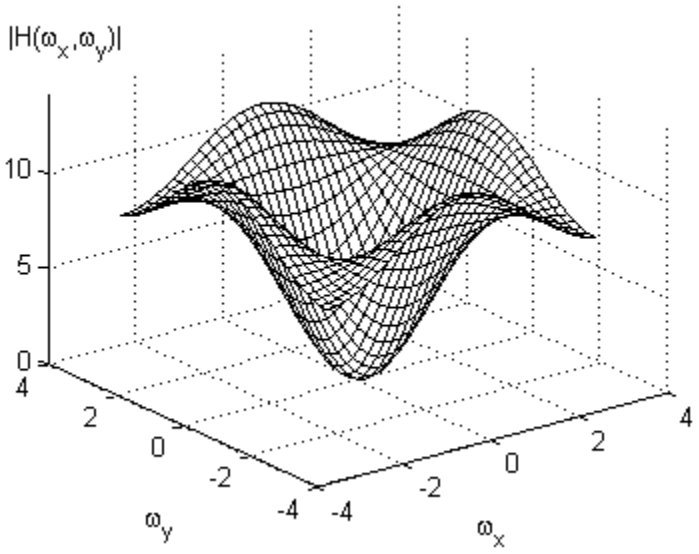
0	1	0
1	-4	1
0	1	0

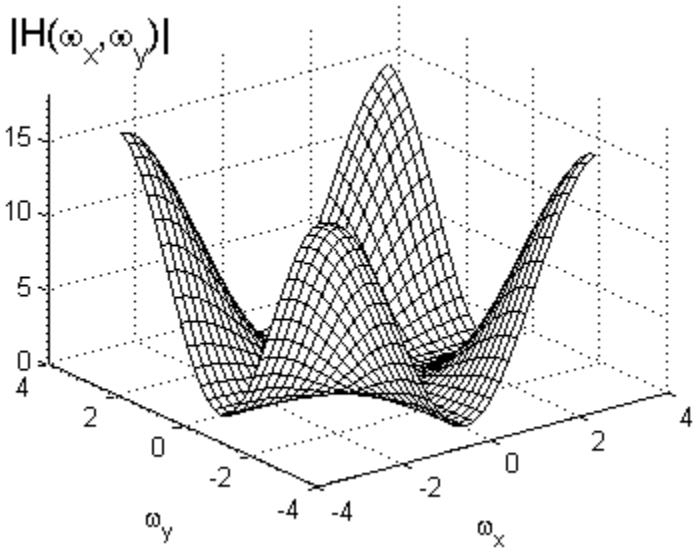
1	1	1
1	-8	1
1	1	1

-1	2	-1
2	-4	2
-1	2	-1

Frequency responses of the highpass filters shown in Fig. 9.


|H(ωx,ωy)|
8
6
4
2
0
4
2
0
-2
-4
ωy
-4
-2
0
2
4
ωx


|H(ωx,ωy)|
10
5
0
4
2
0
-2
-4
ωy
-4
-2
0
2
4
ωx



An example of highpass filtering is illustrated in [link]. It should be noted from this example that the processed image has enhanced contrast, however it appears more noisy than the original image. Since noise will usually contribute to the high frequency content of an image, highpass filtering has the undesirable effect of accentuating the noise.

(1) Original gray scale image. (2) Highpass filtered image.

Sharpening Exercise

Download the file blur.tif for the following section.

In this section, we will introduce a sharpening filter known as an **unsharp mask**. This type of filter subtracts out the "unsharp" (low frequency) components of the image, and consequently produces an image with a sharper appearance. Thus, the unsharp mask is closely related to highpass filtering. The process of unsharp masking an image $f(i,j)$ can be expressed by

Equation:

$$g(i,j) = \alpha f(i,j) - \beta[f(i,j)^{**}\ h(i,j)]$$

where $h(i,j)$ is a **lowpass** filter, and α and β are positive constants such that $\alpha - \beta = 1$.

Analytically calculate the frequency response of the unsharp mask filter in terms of α, β, and $h(i,j)$ by finding an expression for

Equation:

$$\frac{G(\omega_1, \omega_2)}{F(\omega_1, \omega_2)} \ .$$

Using your `gaussFilter` function from the "Smoothing Exercise" section, create a 5×5 Gaussian filter with $\sigma^2 = 1$. Use Matlab to compute the frequency response of an unsharp mask filter (use your expression for [link]), using the Gaussian filter as $h(i,j)$, $\alpha = 5$ and $\beta = 4$. The size of the calculated frequency response should be 32×32. Plot the magnitude of this response in the range $[-\pi, \pi] \times [-\pi, \pi]$ using `mesh`, and label the axes. You can change the viewing angle of the mesh plot with the `view` command. Print out this response.

Download the image file blur.tif and read it into Matlab. Apply the unsharp mask filter with the parameters specified above to this image, using [link].

Use `image` to view the original and processed images. What effect did the filtering have on the image? Label the processed image and print it out.

Now try applying the filter to **blur.tif**, using $\alpha = 10$ and $\beta = 9$. Compare this result to the previous one. Label the processed image and print it out.

INLAB REPORT

1. Hand in your derivation for the frequency response of the unsharp mask.
2. Hand in the labeled plot of the magnitude response. Compare this plot to the highpass responses of [link]. In what ways is it similar to these frequency responses?
3. Hand in the two processed images.
4. Describe any positive and negative effects of the filtering that you observe. Discuss the influence of the α and β parameters.

Lab 10b - Image Processing (part 2)

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

Introduction

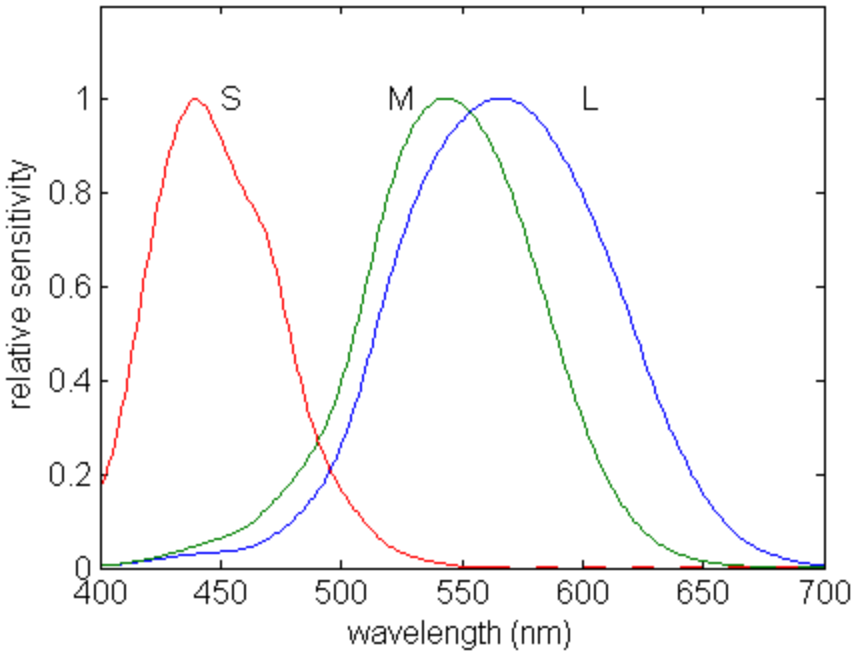
This is the second part of a two week experiment in image processing. In the first week , we covered the fundamentals of digital monochrome images, intensity histograms, pointwise transformations, gamma correction, and image enhancement based on filtering.

During this week, we will cover some fundamental concepts of color images. This will include a brief description on how humans perceive color, followed by descriptions of two standard **color spaces**. We will also discuss an application known as **halftoning**, which is the process of converting a gray scale image into a binary image.

Color Images

Background on Color

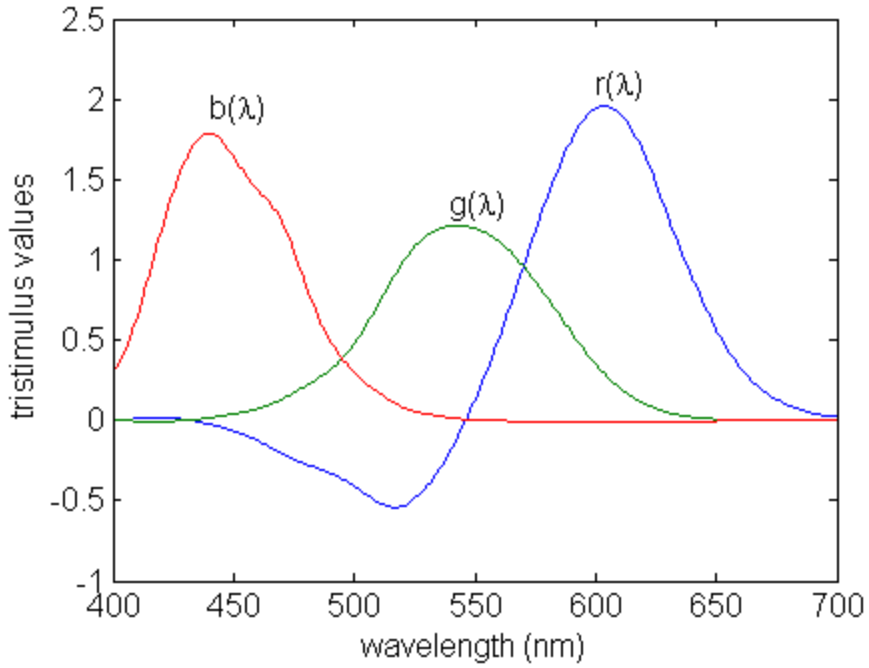
Color is a perceptual phenomenon related to the human response to different wavelengths of light, mainly in the region of 400 to 700 nanometers (nm). The perception of color arises from the sensitivities of three types of neurochemical sensors in the retina, known as the **long** (L), **medium** (M), and **short** (S) **cones**. The response of these sensors to photons is shown in [link]. Note that each sensor responds to a range of wavelengths.



Relative photon sensitivity of long (L), medium (M), and short (S) cones.

Due to this property of the human visual system, all colors can be modeled as combinations of the three **primary color** components: red (R), green (G), and blue (B). For the purpose of standardization, the CIE (Commission International de l'Eclairage — the International Commission on Illumination) designated the following wavelength values for the three primary colors: blue = $435.8nm$, green = $546.1nm$, and red = $700nm$.

The relative amounts of the three primary colors of light required to produce a color of a given wavelength are called **tristimulus values**. [link] shows the plot of tristimulus values using the CIE primary colors. Notice that some of the tristimulus values are **negative**, which indicates that colors at those wavelengths cannot be reproduced by the CIE primary colors.



Plot of tristimulus values using CIE primary colors.

Color Spaces

A **color space** allows us to represent all the colors perceived by human beings. We previously noted that weighted combinations of stimuli at three wavelengths are sufficient to describe all the colors we perceive. These wavelengths form a natural basis, or coordinate system, from which the color measurement process can be described. In this lab, we will examine two common color spaces: RGB and YC_bC_r. For more information, refer to [link].

- **RGB** space is one of the most popular color spaces, and is based on the tristimulus theory of human vision, as described above. The RGB space is a hardware-oriented model, and is thus primarily used in computer monitors and other raster devices. Based upon this color

space, each pixel of a digital color image has three components: red, green, and blue.

- YC_bC_r space is another important color space model. This is a gamma corrected space defined by the CCIR (International Radio Consultative Committee), and is mainly used in the digital video paradigm. This space consists of **luminance** (Y) and **chrominance** (C_bC_r) components. The importance of the YC_bC_r space comes from the fact that the human visual system perceives a color stimulus in terms of luminance and chrominance attributes, rather than in terms of R, G, and B values. The relation between YC_bC_r space and gamma corrected RGB space is given by the following linear transformation.
  Equation:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ C_b &= 0.564(B - Y) + 128 \\ C_r &= 0.713(R - Y) + 128 \end{aligned}$$

In YC_bC_r, the luminance parameter is related to an overall intensity of the image. The chrominance components are a measure of the relative intensities of the blue and red components. The inverse of the transformation in [link] can easily be shown to be the following.
Equation:

$$\begin{aligned} R &= Y + 1.4025(C_r - 128) \\ G &= Y - 0.3443\,(C_b - 128) - 0.7144^*(C_r - 128) \\ B &= Y + 1.7730(C_b - 128) \end{aligned}$$

Color Exercise

Download the files girl.tif and ycbcr.mat. For help on image command select the link.

You will be displaying both color and monochrome images in the following exercises. Matlab's `image` command can be used for both image types, but care must be taken for the command to work properly. Please see the help on the image command for details.

Download the RGB color image file girl.tif , and load it into Matlab using the `imread` command. Check the size of the Matlab array for this image by typing `whos`. Notice that this is a three dimensional array of type `uint8`. It contains three gray scale image planes corresponding to the red, green, and blue components for each pixel. Since each color pixel is represented by three bytes, this is commonly known as a 24-bit image. Display the color image using

```
image(A);
```

```
axis('image');
```

where A is the 3-D RGB array.

You can extract each of the color components using the following commands.

`RGB = imread('girl.tif'); %` color image is loaded into matrix RGB

`R = RGB(:,:,1); %` extract red component from RGB

`G = RGB(:,:,2); %` extract green component from RGB

`B = RGB(:,:,3); %` extract blue component from RGB

Use the `subplot` and `image` commands to plot the original image, along with each of the three color components. Note that while the original is a color image, each color component separately is a monochrome image. Use the syntax `subplot(2,2,n)` , where $n = 1, 2, 3, 4$, to place the four images in the same figure. Place a title on each of the images, and print the figure (use a color printer).

We will now examine the YC_bC_r color space representation. Download the file ycbcr.mat , and load it into Matlab using `load ycbcr`. This file contains a Matlab array for a color image in YC_bC_r format. The array contains three gray scale image planes that correspond to the **luminance** (Y) and two **chrominance** (C_bC_r) components. Use `subplot(3,1,n)` and `image` to display each of the components in the same figure. Place a title on each of the three monochrome images, and print the figure.

In order to properly display this color image, we need to convert it to RGB format. Write a Matlab function that will perform the transformation of [link]. It should accept a 3-D YC_bC_r image array as input, and return a 3-D RGB image array.

Now, convert the **ycbcr** array to an RGB representation and display the color image. Remember to convert the result to type `uint8` before using the `image` command.

An interesting property of the human visual system, with respect to the YC_bC_r color space, is that we are much more sensitive to distortion in the luminance component than in the chrominance components. To illustrate this, we will smooth each of these components with a Gaussian filter and view the results.

You may have noticed when you loaded **ycbcr.mat** into Matlab that you also loaded a 5×5 matrix, h. This is a 5×5 Gaussian filter with $\sigma^2 = 2.0$. (See the first week of the experiment for more details on this type of filter.) Alter the **ycbcr** array by filtering only the luminance component, `ycbcr(:,:,1)` , using the Gaussian filter (use the `filter2` function). Convert the result to RGB, and display it using `image`. Now alter **ycbcr** by filtering both chrominance components, `ycbcr(:,:,2)` and `ycbcr(:,:,3)`, using the Gaussian filter. Convert this result to RGB, and display it using `image`.

Use `subplot(3,1,n)` to place the original and two filtered versions of the **ycbcr** image in the same figure. Place a title on each of the images, and print the figure (in color). Do you see a significant difference between the filtered versions and the original image? This is the reason that YC_bC_r is

often used for digital video. Since we are not very sensitive to corruption of the chrominance components, we can afford to lose some information in the encoding process.

INLAB REPORT

1. Submit the figure containing the components of **girl.tif**.
2. Submit the figure containing the components of **ycbcr**.
3. Submit your code for the transformation from YC_bC_r to RGB.
4. Submit the figure containing the original and filtered versions of **ycbcr**. Comment on the result of filtering the luminance and chrominance components of this image. Based on this, what conclusion can you draw about the human visual system?

Halftoning

In this section, we will cover a useful image processing technique called **halftoning**. The process of halftoning is required in many present day electronic applications such as facsimile (FAX), electronic scanning/copying, laser printing, and low bandwidth remote sensing.

Binary Images

As was discussed in the first week of this lab, an 8-bit monochrome image allows 256 distinct gray levels. Such images can be displayed on a computer monitor if the hardware supports the required number intensity levels. However, some output devices print or display images with much fewer gray levels. In the extreme case, the gray scale images must be converted to binary images, where pixels can only be black or white.

The simplest way of converting to a binary image is based on **thresholding**, i.e. two-level (one-bit) quantization. Let $f(i,j)$ be a gray scale image, and $b(i,j)$ be the corresponding binary image based on thresholding. For a given threshold T, the binary image is computed as the following:
Equation:

$$b\left(i,j\right) = \begin{cases} 255 & \text{if } f(i,j) > T \\ 0 & \text{else} \end{cases}$$

(1) Original gray scale image. (2) Binary image produced by simple fixed thresholding.

[link] shows an example of conversion to a binary image via thresholding, using $T = 80$.

It can be seen in [link] that the binary image is not "shaded" properly–an artifact known as **false contouring**. False contouring occurs when quantizing at low bit rates (one bit in this case) because the quantization error is dependent upon the input signal. If one reduces this dependence, the visual quality of the binary image is usually enhanced.

One method of reducing the signal dependence on the quantization error is to add uniformly distributed white noise to the input image prior to quantization. To each pixel of the gray scale image $f(i, j)$, a white random number n in the range $[-A, A]$ is added, and then the resulting image is quantized by a one-bit quantizer, as in [link]. The result of this method is illustrated in [link], where the additive noise is uniform over $[-40, 40]$. Notice that even though the resulting binary image is somewhat noisy, the false contouring has been significantly reduced.

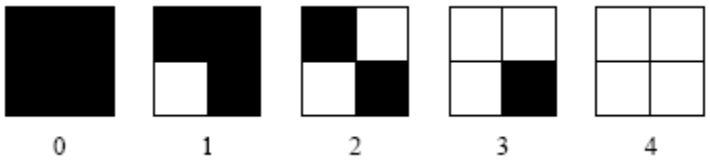
Note: Depending on the pixel size, you sometimes need to view a halftoned image from a short distance to appreciate the effect. The natural filtering (blurring) of the visual system allows you to perceive many different shades, even though the only colors displayed are black and white!

Random noise binarization.

Ordered Dithering

Halftone images are binary images that appear to have a gray scale rendition. Although the random thresholding technique described in "Binary Images" can be used to produce a halftone image, it is not often used in real applications since it yields very noisy results. In this section, we will describe a better halftoning technique known as **ordered dithering**.

The human visual system tends to average a region around a pixel instead of treating each pixel individually, thus it is possible to create the illusion of many gray levels in a binary image, even though there are actually only two gray levels. With 2×2 binary pixel grids, we can represent 5 different "effective" intensity levels, as shown in [link]. Similarly for 3×3 grids, we can represent 10 distinct gray levels. In dithering, we replace blocks of the original image with these types of binary grid patterns.



Five different patterns of 2×2 binary pixel grids.

Remember from "Binary Images" that false contouring artifacts can be reduced if we can reduce the signal dependence or the quantization error. We showed that adding uniform noise to the monochrome image can be used to achieve this decorrelation. An alternative method would be to use a variable threshold value for the quantization process.

Ordered dithering consists of comparing blocks of the original image to a 2-D grid, known as a **dither pattern**. Each element of the block is then quantized using the corresponding value in the dither pattern as a threshold. The values in the dither matrix are fixed, but are typically different from each other. Because the threshold value varies between adjacent pixels, some decorrelation from the quantization error is achieved, which has the effect of reducing false contouring.

The following is an example of a 2×2 dither matrix,
Equation:

$$T\left(i,j\right)=255*\begin{bmatrix}5/8 & 3/8\\1/8 & 7/8\end{bmatrix}$$

This is a part of a general class of optimum dither patterns known as **Bayer matrices**. The values of the threshold matrix $T(i,j)$ are determined by the order that pixels turn "ON". The order can be put in the form of an **index matrix**. For a Bayer matrix of size 2, the index matrix $I(i,j)$ is given by
Equation:

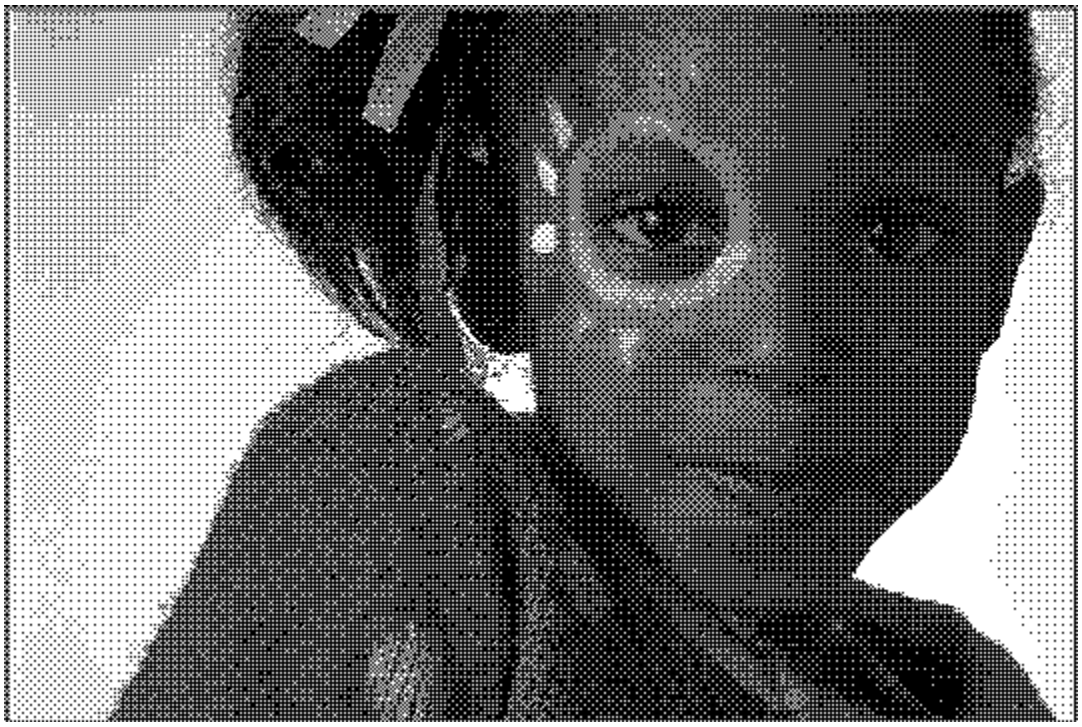
$$I\left(i,j\right)=\begin{bmatrix}3 & 2\\1 & 4\end{bmatrix}$$

and the relation between $T(i,j)$ and $I(i,j)$ is given by
Equation:

$$T\left(i,j\right)=255\left(I\left(i,j\right)-0.5\right)/n^2$$

where n^2 is the total number of elements in the matrix.

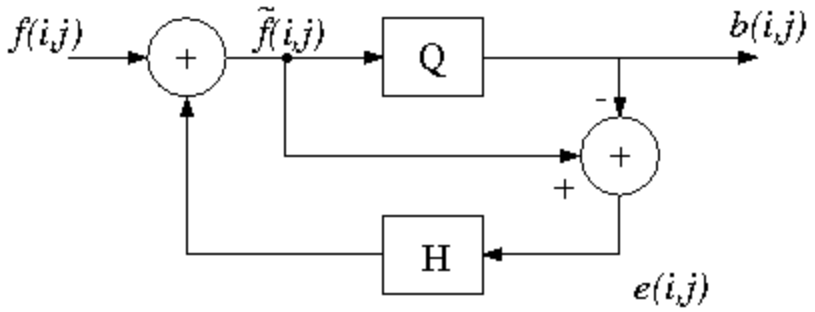
[link] shows the halftone image produced by Bayer dithering of size 4. It is clear from the figure that the halftone image provides good detail rendition. However the inherent square grid patterns are visible in the halftone image.



The halftone image produced by Bayer dithering of size 4.

Error Diffusion

Another method for halftoning is random dithering by **error diffusion**. In this case, the pixels are quantized in a specific order (raster ordering[footnote] is commonly used), and the residual quantization error for the current pixel is propagated (diffused) forward to unquantized pixels. This keeps the overall intensity of the output binary image closer to the input gray scale intensity.
Raster ordering of an image orients the pixels from left to right, and then top to bottom. This is similar to the order that a CRT scans the electron beam across the screen.



Block diagram of the error diffusion method.

[link] is a block diagram that illustrates the method of error diffusion. The current input pixel $f(i,j)$ is modified by means of past quantization errors to give a modified input $\widetilde{f}(i,j)$. This pixel is then quantized to a binary value by Q, using some threshold T. The error $e(i,j)$ is defined as
Equation:

$$e(i,j) = \widetilde{f}(i,j) - b(i,j)$$

where $b(i,j)$ is the quantized binary image.

The error $e(i,j)$ of quantizing the current pixel is is diffused to "future" pixels by means of a two-dimensional weighting filter $h(i,j)$, known as the **diffusion filter**. The process of modifying an input pixel by past errors can be represented by the following recursive relationship.
Equation:

$$\widetilde{f}(i,j) = f(i,j) + \sum_{k,l \in S} h(k,l) e(i-k, j-l)$$

The most popular error diffusion method, proposed by Floyd and Steinberg, uses the diffusion filter shown in [link]. Since the filter coefficients sum to one, the local average value of the quantized image is equal to the local average gray scale value. [link] shows the halftone image produced by

Floyd and Steinberg error diffusion. Compared to the ordered dither halftoning, the error diffusion method can be seen to have better contrast performance. However, it can be seen in [link] that error diffusion tends to create "streaking" artifacts, known as **worm** patterns.

	•	7/16
3/16	5/16	1/16

The error diffusion filter proposed by Floyd and Steinberg.

A halftone image produced by the Floyd and Steinberg error diffusion method.

Halftoning Exercise

Download the file house.tif for the following section.

Note: If your display software (e.g Matlab) resizes your image before rendering, a halftoned image will probably not be rendered properly. For example, a subsampling filter will result in gray pixels in the displayed image! To prevent this in Matlab, use the

```
truesize
```

command just after the

`image`

command. This will assign one monitor pixel for each image pixel.

We will now implement the halftoning techniques described above. Save the result of each method in MAT files so that you may later analyze and compare their performance. Download the image file house.tif and read it into Matlab. Print out a copy of this image.

First try the simple thresholding technique based on [link], using $T = 108$, and display the result. In Matlab, an easy way to threshold an image X is to use the command `Y = 255*(X>T);` . Label the quantized image, and print it out.

Now create an "absolute error" image by subtracting the binary from the original image, and then taking the absolute value. The degree to which the original image is present in the error image is a measure of signal dependence of the quantization error. Label and print out the error image.

Compute the mean square error (MSE), which is defined by
Equation:

$$MSE = \frac{1}{NM} \sum_{i,j} \{f(i,j) - b(i,j)\}^2$$

where NM is the total number of pixels in each image. Note the MSE on the printout of the quantized image.

Now try implementing Bayer dithering of size 4. You will first have to compute the dither pattern. The index matrix for a dither pattern of size 4 is given by
Equation:

$$I(i,j) = \begin{matrix} 12 & 8 & 10 & 6 \\ 4 & 16 & 2 & 14 \\ 9 & 5 & 11 & 7 \\ 1 & 13 & 3 & 15 \end{matrix}.$$

Based on this index matrix and [link], create the corresponding threshold matrix.

For ordered dithering, it is easiest to perform the thresholding of the image all at once. This can be done by creating a large threshold matrix by repeating the 4×4 dither pattern. For example, the command `T = [T T; T T];` will increase the dimensions of T by 2. If this is repeated until T is at least as large as the original image, T can then be trimmed so that is is the same size as the image. The thresholding can then be performed using the command `Y = 255*(X>T);` .

As above, compute an error image and calculate the MSE. Print out the quantized image, the error image, and note the MSE.

Now try halftoning via the error diffusion technique, using a threshold $T = 108$ and the diffusion filter in [link]. It is most straightforward to implement this by performing the following steps on each pixel in raster order:

1. Initialize an output image matrix with zeros.
2. Quantize the current pixel using using the threshold T, and place the result in the output matrix.
3. Compute the quantization error by subtracting the binary pixel from the gray scale pixel.
4. Add scaled versions of this error to "future" pixels of the original image, as depicted by the diffusion filter of [link].
5. Move on to the next pixel.

You do not have to quantize the outer border of the image.

As above, compute an error image and calculate the MSE. Print out the quantized image, the error image, and note the MSE.

The human visual system naturally lowpass filters halftone images. To analyze this phenomenon, filter each of the halftone images with the Gaussian lowpass filter h that you loaded in the previous section (from **ycbcr.mat**), and measure the MSE of the filtered versions. Make a table that contains the MSE's for both filtered and nonfiltered halftone images for each of the three methods. Does lowpass filtering reduce the MSE for each method?

INLAB REPORT

1. Hand in the original image and the three binary images. Make sure that they are all labeled, and that the mean square errors are noted on the binary images.
2. Compare the performance of the three methods based on the visual quality of the halftoned images. Also compare the resultant MSE's. Is the MSE consistent with the visual quality?
3. Submit the three error images. Which method appears to be the least signal dependent? Does the signal dependence seem to be correlated with the visual quality?
4. Compare the MSE's of the filtered versions with the nonfiltered versions for each method. What is the implication of these observations with respect to how we perceive halftone images.